NAVAL POSTGRADUATE SCHOOL
Monterey, California

THESIS

PARALLEL PROCESSING OF NAVY SPECIFIC APPLICATIONS USING A WORKSTATION CLUSTER

by

Leon Conrad Stone, Jr.

December 1993

Thesis Advisor: Shridhar Shukla
Co-Advisor: Beny Neta

Approved for public release; distribution is unlimited.

REPORT DOCUMENTATION PAGE

Form Approved OMB No. 0704-0188

1. AGENCY USE ONLY *(Leave blank)*	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	16 December 1993.	Master's Thesis

4. TITLE AND SUBTITLE PARALLEL PROCESSING OF NAVY SPECIFIC APPLICATIONS USING A WORKSTATION CLUSTER.

5. FUNDING NUMBERS

6. AUTHOR(S) Leon Conrad Stone Jr.

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Naval Postgraduate School
Monterey CA 93943-5000

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government.

12a. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution is unlimited.

12b. DISTRIBUTION CODE
A

13. ABSTRACT *(maximum 200 words)*

In this thesis the benefits of parallel computing using a workstation cluster are explored for two typical Naval applications. The applications are examples of one off-line and one on-line program. The off-line program is a Navy program currently in use by the Naval Space Command in its satellite prediction model. The on-line program is a large grain data flow problem with critical throughput requirements and represents a hypothetical combat weapons system. Data and function decomposition techniques are used in both applications. Speedup and throughput are the performance metrics studied.

The software employed was the Parallel Virtual Machine (PVM) by the Oak Ridge National Laboratory. PVM enables a network of heterogeneous workstations to appear as a parallel multicomputer to the user programs. PVM runs over the workstation operating system and provides the user with a set of library calls for message passing and process creation.

14. SUBJECT TERMS Parallelization, PPT2, PVM, Large Grain Data Flow Problems

15. NUMBER OF PAGES
*** 128

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
Unclassified	Unclassified	Unclassified	UL

NSN 7540-01-280-5500

Standard Form 298 (Rev. 2-89)
Prescribed by ANSI Std. 239-18

Approved for public release; distribution is unlimited.

PARALLEL PROCESSING OF NAVY SPECIFIC APPLICATIONS USING A WORKSTATION CLUSTER

by

Leon Conrad Stone Jr.

Lieutenant, United States Navy
B.S., Purdue University, 1987

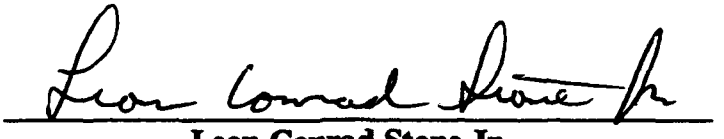
Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN ELECTRICAL ENGINEERING

from the

NAVAL POSTGRADUATE SCHOOL

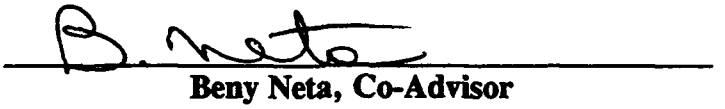
December 1993

Author: 

Leon Conrad Stone Jr.

Approved by: 

Shridhar Shukla, Thesis Advisor



Beny Neta, Co-Advisor

Michael A. Morgan, Chairman,
Department of Electrical and Computer Engineering

ABSTRACT

In this thesis the benefits of parallel computing using a workstation cluster are explored for two typical Naval applications. The applications are examples of one off-line and one on-line program. The off-line program is a Navy program currently in use by the Naval Space Command in its satellite prediction model. The on-line program is a large grain data flow problem with critical throughput requirements and represents a hypothetical combat weapons system. Data and function decomposition techniques are used in both applications. Speedup and throughput are the performance metrics studied.

The software employed was the Parallel Virtual Machine (PVM) by the Oak Ridge National Laboratory. PVM enables a network of heterogeneous workstations to appear as a parallel multicomputer to the user programs. PVM runs over the workstation operating system and provides the user with a set of library calls for message passing and process creation.

TABLE OF CONTENTS

LIST OF FIGURES

ACKNOWLEDGMENT

I wish to express my deep appreciation for my advisor, Professor Shridhar Shukla for his tireless support and his ceaseless confidence in my abilities throughout this research. To my co-advisor, Professor Beny Neta, I wish to thank for bringing the PPT2 opportunity to my attention and for his appreciation of my endeavors.

Finally, I wish to thank my wife and daughter, Sue and Julia. Their unconditional love, unwavering patience, and, most of all, their total understanding made this thesis possible.

I. INTRODUCTION

All successful organizations depend on reliable and timely data management. As an organization evolves, its data system requirements also increase. The United States Navy is an example of one such organization. Its data processing requirements demand evermore computing speed and capacity. An economical solution to this need is to network the workstations present in abundance and utilize parallel processing. To this end, this thesis provides performance results of two typical applications on a workstation cluster.

With the introduction of small, relatively inexpensive computers, a vast amount of computing resources are often left idle for a long period of time. A ship often has this characteristic. A ship's complement of computers is usually used for intermittent word processing or single dedicated computational tasks. With these computers networked together, a lot of unused CPU power is available. In order to tap into these unused assets, parallelization software tools have been developed. These programs operate at the user level like an extra layer of operating system code.

The Navy's computation requirements can be classified as off-line and on-line data processing programs. An off-line program does not require continuous, time-critical, processing. It executes once per some specified time period with clear beginning and ending times. An on-line program does require continuous computational assets for its functions. It is characterized by constant, non-stop, real time processing .

For this thesis, one example of each type of program was parallelized using a software tool. The tool used for parallelization was the Parallel Virtual Machine (PVM). The off-line program was the Naval Space Command's PPT2 Analytical

Satellite Position Propagation Program. The on-line program is a hypothetical Shipboard Combat Weapon System.

A. PVM: PARALLEL VIRTUAL MACHINE

PVM is a software library, currently being refined, developed by the Oak Ridge National Laboratory (ORNL). It is a software system that enables a collection of heterogeneous computers to be used as a coherent and flexible concurrent computational system [Ref. 1]. PVM was chosen because it is relatively easy to use, is an emerging standard for software of its kind, and its price is definitely reasonable. It is currently available free of charge from ORNL and installation is relatively easy. PVM version 3.2 was used for this thesis. A short description on acquiring and installing PVM appears in Appendix A.

B. THESIS SCOPE AND CONTRIBUTION

The goal of this thesis was to exploit the benefits of parallel computing, as cost effectively as possible, using a software tool. The two applications process large amounts of data and represent contrasting requirements while lending themselves to parallel processing. The positive aspect of parallelizing these procedures is the performance improvement over their serial counterparts. Parallelization could have been accomplished using a specific parallel multicomputer. These systems tend to be large and expensive, and tie-up extensive human and fiscal resources for a limited number of uses. PVM provided the desired cost effectiveness. While, arguably, PVM may not accomplish the tasks as fast as, say, an INTEL iPSC/2 hypercube, the process execution times were satisfactory for the applications tested. Furthermore, they were accomplished on a shared network without noticeably disturbing other system users.

The representative use of a loosely shared network in this thesis is the most noteworthy aspect. For instance, the on-line application was tested as though it was in a shipboard environment. It performed as desired while simulated shipboard tasks such as, supply data-base upkeep, report and correspondence word processing, and computerized engineering parameter measuring were being carried out.

C. THESIS ORGANIZATION

This thesis is organized as follows. Chapters II and III cover the Naval Space Command's PPT2 program. Chapter II specifically describes PPT2 itself, the modes of parallelization used, and the variable mode which was finally used. Chapter III reports the results obtained for this application and recommendations for possible future improvements to the model.

Chapters IV and V deal with the hypothetical Combat System model. Chapter IV details the design and requirements of the model and Chapter V contains the results.

Chapter VI contains overall conclusions and areas for further study.

II. PARALLELIZATION OF PPT2

Currently the Naval Space Command tracks over 6000 objects orbiting around the earth. With more and more countries entering space exploitation, and as the United States increases its emphasis on space communication, this data set of satellites will forseeably increase dramatically in the future. These increases in the satellite catalog will increase the computational demands on the computer tasked with orbit prediction. If the NAVSPACECOM's orbital model's accuracy is increased or multiple calls to the orbit prediction algorithm are made for accuracy, the computational demands may be too much of a burden if the computer was a serial machine [Ref. 2]. Given these computational loads, and the time dependency of the results, parallel processing of the catalog is a logical extension.

A. PPT2

PPT2 is the NAVSPACECOM's program which implements an analytic satellite motion model based on the Brouwer-Lyddane orbital prediction theory. Reference [2] goes into great depth describing this theory and how PPT2 implements the theory in FORTRAN. For this thesis, the accuracy of the PPT2 program, or the theory of how it works was not relevant. The one major aspect of PPT2 considered was the required size of each satellite data record which is 84 elements. No other internal details of PPT2 are discussed here.

B. PARALLEL DECOMPOSITION METHODS

Given a program and its associated data set, there are two primary ways to process it in parallel. The program can be separated into individual sections with a processor dedicated to compute its respective part, much like a factory assembly line. The other

primary method is dividing up the data set and sending parts to many separate processors all running the same algorithm, but on different data. Each of these methods is highly dependent on the program description and the size of the data.

Although the PPT2 algorithm is sufficiently large to break down into individual computational nodes, the data set size is such that data decomposition is more effective. These observations are validated in Reference [2]. Control decomposition had been previously attempted but was not successful [Ref. 3]. Based on these results, all of the parallelization methods used were various ways of decomposing the satellite catalogue and distributing it to multiple nodes executing PPT2.

C. DECOMPOSITION STRATEGIES

The basic algorithm for all of the decomposition strategies used a master/slave distribution network. For all the programs, there was one supervisor (master) node which decomposed the data set and distributed it to the worker (slave) nodes. Each worker ran on a separate processor and sent its results to a gathering node which printed the results to a file and reported to the supervisor when the process had completed for all satellites. Figure 2.1 graphically presents these relationships.

To get a general understanding of the decomposition requirements multiple decomposition strategies were developed, each with benefits over the previous strategy until four different methods had been explored. All the methods endeavored to keep the worker processors busy as much as possible to increase speedup and efficiency. Each method is described below.

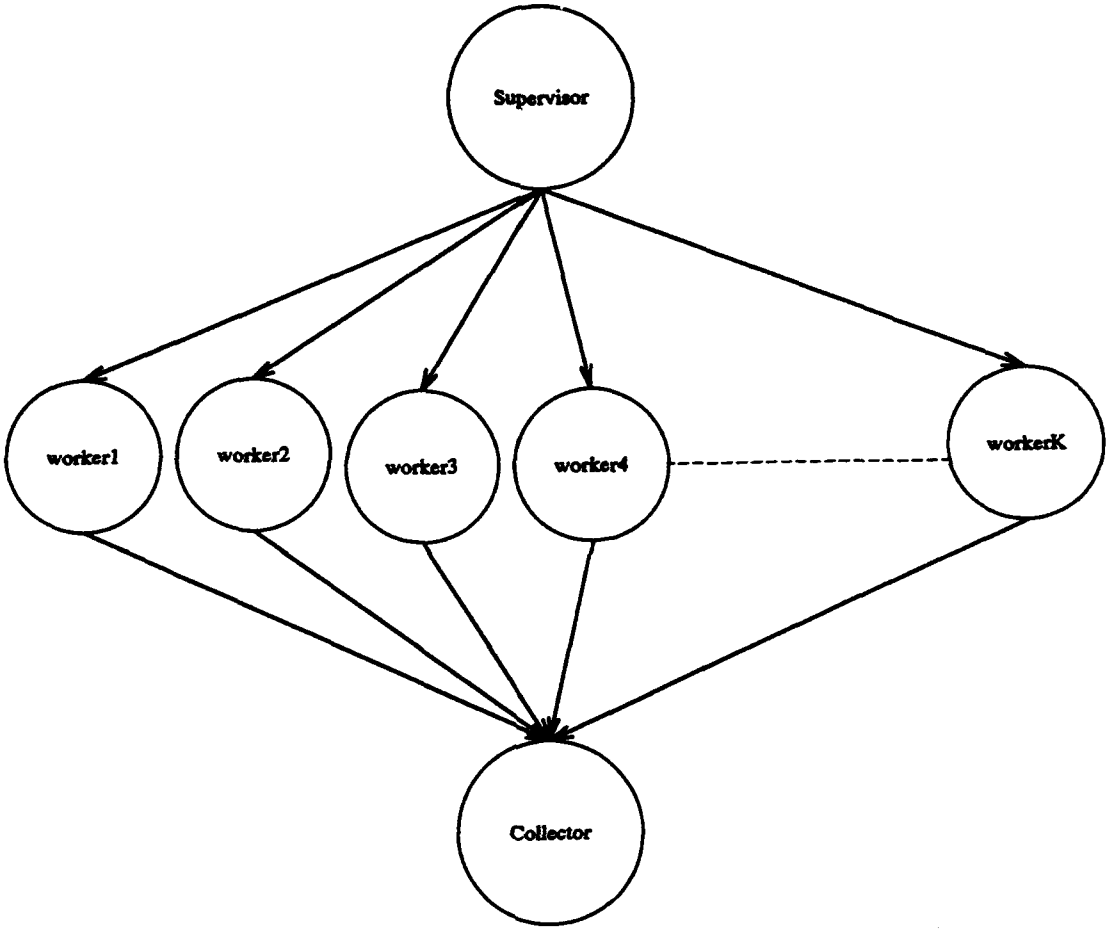


Figure 2.1. Supervisor/Worker Dependency Graph.

1. **ds1: Send/Request One at a Time**

For this strategy, the supervisor node initially sends one satellite to each individual worker node and waits for the workers to individually request another satellite. This method brought out the high PVM communications overhead which needed to be overcome for adequate speedup.

2. **ds2: Send/No Request**

The supervisor node for this routine sent one satellite at a time to each worker node until the input file was exhausted. This process reduced the communications overhead between the supervisor and worker, but it did not keep all the processors busy for a sufficiently long time.

3. **ds3: Send Block**

For this scheme, the supervisor divided the number, S, of input satellites by the number, n, of worker processors. The supervisor then decomposed the input data into blocks of S/n size and distributed these to each processor individually. This was much more efficient than the previous two methods, but for a large n, $n > 8$, the workers numbered eight and above were still not getting data fast enough to notice effective processor computational overlap.

4. **ds4: Send Half Block**

For this scheme, the supervisor divided the S/n size block by two then sent the two half blocks to each worker so all the workers had one half of their data while the supervisor was sending the second half. These schemes were used with data sets of 600 and 1200 satellites. For experimentation, PVM was started on eighteen different workstations so measurements could be taken for one to sixteen working nodes.

Figure 2.2. PVM Applied to PPT2 Using 600 Input Satellites.

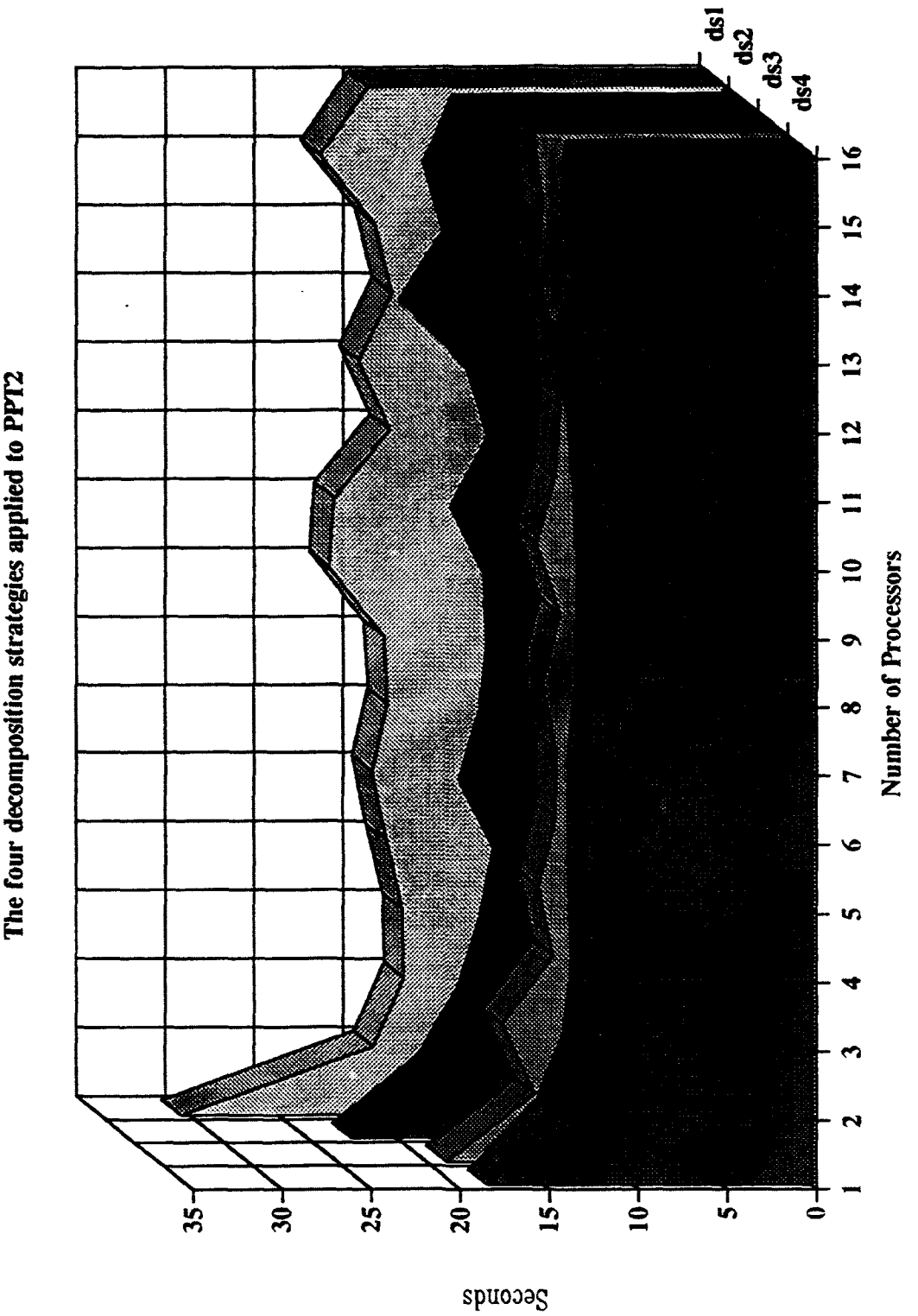


Figure 2.3. PVM Applied to PPT2 Using 1200 Input Satellites.

The collected data consisted of the actual execution time taken to process all the elements in the input files. The programs were run ten times for each number of processors in order to get a good average time. They were executed at times when the network was minimally used to avoid, as much as possible, bus contentions with other users. The results of these are given in Figures 2.2 and 2.3. These figures show a definite advantage in sending two input blocks of data to each worker node over the other schemes.

Some other decomposition strategies were experimented with, but not in as much detail. One strategy was to send the entire input data to all of the workers simultaneously and let the worker nodes extract the data they were to use. This method was memory prohibitive and its execution time was about the same as ds2 from Figures 2.2 and 2.3. Other data distribution techniques involved various methods of packing and unpacking the data to be sent via PVM. Only the data block decomposition schemes could take advantage of these attempts, but the execution time improvements were slight.

D. MULTIPLE BLOCK DECOMPOSITION SCHEME: DS5

The data decomposition scheme ds4 was modified to send a variety of block sizes depending on the size of the input and the number of working nodes used. In this scheme, ds5, the supervisor still sent a block of data to each worker, then the worker extracted one satellite at a time from its input buffer and sent a block of results, equal in size to its input block, to the gathering node. The FORTRAN code for the ds5 supervisor and worker/gathering nodes is in Appendix B. In PVM, the buffer manipulation time is the costliest aspect of communications which is why this scheme optimized the performance. Sending blocks of data between processors vice one data

element at a time, minimized the buffer manipulation which resulted in lower execution times for this data distribution scheme.

The next chapter provides the results of using this scheme. Theoretical execution time equations were developed for this scheme and compared to the actual results. The optimal number of processors and number of input blocks to use were also calculated along with values for speedup.

III. RESULTS OF PPT2 WITH PVM

The results presented in this chapter were obtained using the data block decomposition strategy, ds5, discussed in Chapter II. Eight working nodes were used for all ds5 program runs and were used to obtain the data for all the figures in this chapter. The ds5 supervisor and worker programs were run under PVM, on the Naval Postgraduate School's ECE local area network of various SUN/SPARC workstations. The ECE LAN is an Ethernet based network of various types of workstations. In order to maintain data integrity, only SPARC IPX and SPARC II machines were used. These machines have 40 MHz processors and have been configured with 32 Mbytes of system memory and are essentially the same systems.

A. INITIAL WORKER EXECUTION TIME EQUATION DERIVATIONS

To determine the length of time required to run the parallel algorithm, ds5, the execution time of each working node needed to be determined. This execution time was broken down into three phases: setup, calculation, and breakdown. During the setup phase the worker node waited for and received the next input block from the supervisor. The calculation phase is the time it took for PPT2 to execute on the entire input block of data. The breakdown phase was simply the period in which the worker node packed and sent the results to the gathering node.

In order to obtain an expression for the three phase times, certain variables need to be introduced to represent applicable parts of the program process. Table 3.1 contains a list of the basic variables used and their definitions. Using the variables in Table 3.1, expressions for the setup time, t_s, the calculation time, t_c, and the breakdown time, t_b, were derived for the i^{th} worker processor, P_i.

TABLE 3.1. BASIC VARIABLE LIST.

Variable	Definition
S	total number of satellites in the input file
t_o	node process initialization time
t_{gm}	time for gathering node to report to the supervisor the process is complete
n_b	number of blocks sent to each worker
C_f	fixed communications time for buffer setup and network access for sending records
C_{ps}	communications time required to pack and send one satellite record
C_{upf}	fixed communications time to unpack the input buffer
C_{upps}	communications time to unpack one satellite record
k	number of working processors used
S_p	number of satellites sent to each worker = S/k
S_b	number of satellites per data block = S_p/n_b
T_{ppt2}	time for PPT2 to operate on one satellite record

1. Setup Phase Timing Analysis

The time it takes for the ith node to setup is basically dependent on the time it takes for the master to send the data blocks and the time required to unpack the input buffer. Initially, the working node on processor P_i will have to wait for the master to send data blocks to all the workers j, where $j < i$, before the first block is sent to P_i. The time required to send this first block of data, t_{fb}, and the time to unpack each block make up the setup time.

The time required to send the first block is represented by Equation 3.1:

$$t_{fb} = i \cdot t_{1b} \tag{3.1}$$

where t_{1b} is the time to send one block of data which is the fixed net communications time added to the product of the communications time per satellite and the number of satellites per block is as stated in Equation 3.2.

$$t_{1b} = \left(C_f + C_{ps} \cdot S_b\right) \tag{3.2}$$

The time to unpack the buffer, t_u, is the time spent by P_i to unpack all of the blocks of data. This time is expressed in Equation 3.3.

$$t_u = n_b\left(C_{upf} + C_{upps} S_b\right) \tag{3.3}$$

The total setup time can now be expressed as:

$$t_s = t_{fb} + t_u \tag{3.4}$$

which is simply the sum of the first block communications time and the unpacking time for all of the blocks of data.

2. Calculation Phase Timing Analysis

The calculation time is the time it takes for the PPT2 algorithm to process one block of satellite records. Since t_c is a function of the block size, the equation for the calculation time is:

$$t_c = T_{ppt2} \cdot S_b \tag{3.5}$$

3. Breakdown Phase Timing Analysis

The breakdown phase is the time required for the working node to send one block of results to the gathering node. The expression for t_b is:

$$t_b = \left(C_f + C_{ps} \cdot S_b\right) = t_{1b} \tag{3.6}$$

Using the equations for the three phases and empirical values for the variables, which will be discussed later, the worker's total execution time was determined. The execution times of eight worker nodes, given four input blocks of data, are shown in Figure 3.1. The processor's phase times are described by two lines.

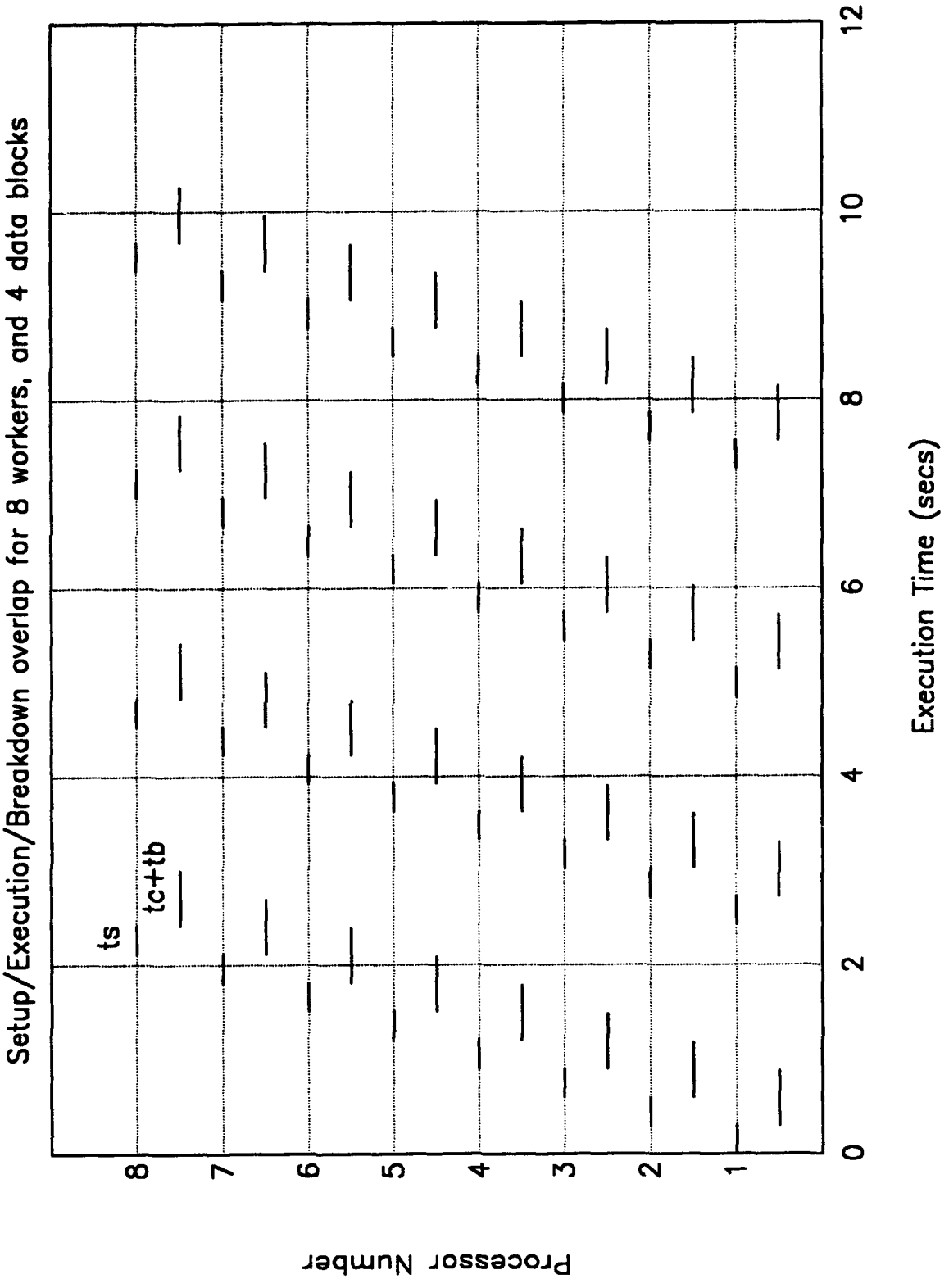


Figure 3.1. ds5 Worker Execution Times Using Eight Worker Processors.

The setup times are the lines on the processor number axis, and the execution and breakdown times are on the line one half space below the processor number. The blank space between the worker's breakdown phase and the next setup time is idle time. This idle time is clearly the result of the communications time required by the master to send blocks to all the working nodes, taking longer than the execution time of PPT2 on each processor. Given the fact PPT2 may need to be run several times for accuracy or tracking requirements, the calculation time needs to be scaled by some constant, A, to take into account multiple iterations. The variable A is the number of times PPT2 is executed on each block of data.

B. EXECUTION TIME EQUATIONS

Looking at Figure 3.1 again, it is clear the worker execution time for the ith worker, for any i, $i = 1,\ldots,k$, is the total setup time added to one calculation and breakdown time. This is true unless the calculation time dominates over the communications time. As a result, instead of a single equation for the total worker execution time, $P_{i\_runtime}$, there are two equations depending on the value of A. Thus, the total time to execute a worker node on processor P_i, or $P_{i\_runtime}$ is determined using Equations 3.7 or 3.9.

The bracketed term in Equation 3.7 is the time in-between the end of a breakdown period and the beginning of the next calculation phase. This time is simply the time required by the supervisor to distribute a data block to all k workers for each block except the first block. The subtraction of the unpack time within the brackets is required because the expression for the setup time is made of n_b unpack times and Equation 3.7 only relies on the unpack time for the final block.

$$P_{i\_runtime} = t_s + A \cdot t_c + t_b + \left[(n_b - 1)(kt_{1b} - t_u / n_b)\right] \tag{3.7}$$

for

$$A \le \frac{(k-1)t_{1b} + \left(C_{upf} + C_{upps} S_b\right)}{t_c} \tag{3.8}$$

and

$$P_{i\_runtime} = t_s + n_b\left(A \cdot t_c + t_b\right) \tag{3.9}$$

for

$$A > \frac{(k-1)t_{1b} + \left(C_{upf} + C_{upps} S_b\right)}{t_c} \tag{3.10}$$

The two expressions for A are taken from Figure 3.1. Equation 3.7 simply means that if the total calculation time and breakdown times are less than the time between setup phases then the communications cost is dominate. Conversely, Equation 3.9 is for the case when the number of iterations of PPT2 causes the calculation phase to dominate.

From the above equations, the total execution time, T_E, of the parallel algorithm is:

$$T_E = t_o + P_{k\_runtime} + t_{gm} \tag{3.11}$$

It should be noted that this equation uses the operation time of the k^{th} worker. The k^{th} worker is used because it is at the end of the data distribution chain and takes longer to complete execution relative to the other workers.

C. PARALLEL AND SERIAL PROGRAM COMPARISON

The comparison of the parallel program vs. serial program entailed theoretical and actual results. In order to accomplish the theoretical comparison, values for the variables in Table 3.1 were needed. Appendix C contains the empirical results from studying the performance of PVM on the ECE SUN network. These values were then

used in the preceding equations for empirical evaluation of the two programs. The total execution time of the serial program was taken to be simply T_{ppt2} multiplied by the total number of satellites in the input file. Again, input and output times were assumed to have been roughly equal for both programs so they were left out of the evaluations.

The input file used for testing consisted of the same satellite records used in Reference [2]. This data file consisted of ten different records which were then duplicated for a total of 4800 input records. An unclassified copy of a portion of the catalog was obtained from the Naval Space Command after the research was completed, and was not used for program development or testing.

Figure 3.2 show the final comparative results. The theoretical lines refer to using Equation 3.11. The actual lines represent data obtained from running the serial program and ds5, (utilizing 8 workers), using values of A from 1 to 10 for both programs. A block size of four was also used for the parallel algorithm. Figure 3.2 shows the parallel program performed better than the serial program as the number of calls to PPT2 was increased. This performance improvement was predicted from the theoretical plots even though the actual serial program performed better than expected and the actual parallel program performed slightly worse than expected. It can also be noted that when $A \approx 7$, Equation 3.11 switches from using Equation 3.7 for the worker processor run time to Equation 3.9. The most dramatic event this figure displays is the fact the parallel program did not perform as well as the serial program for $A = 1$. Since one of the assumptions of this research was the fact PPT2 will most likely be executed a multiple of times, the results for the case $A = 1$ are to be noted but should not detract from the benefits of parallelization.

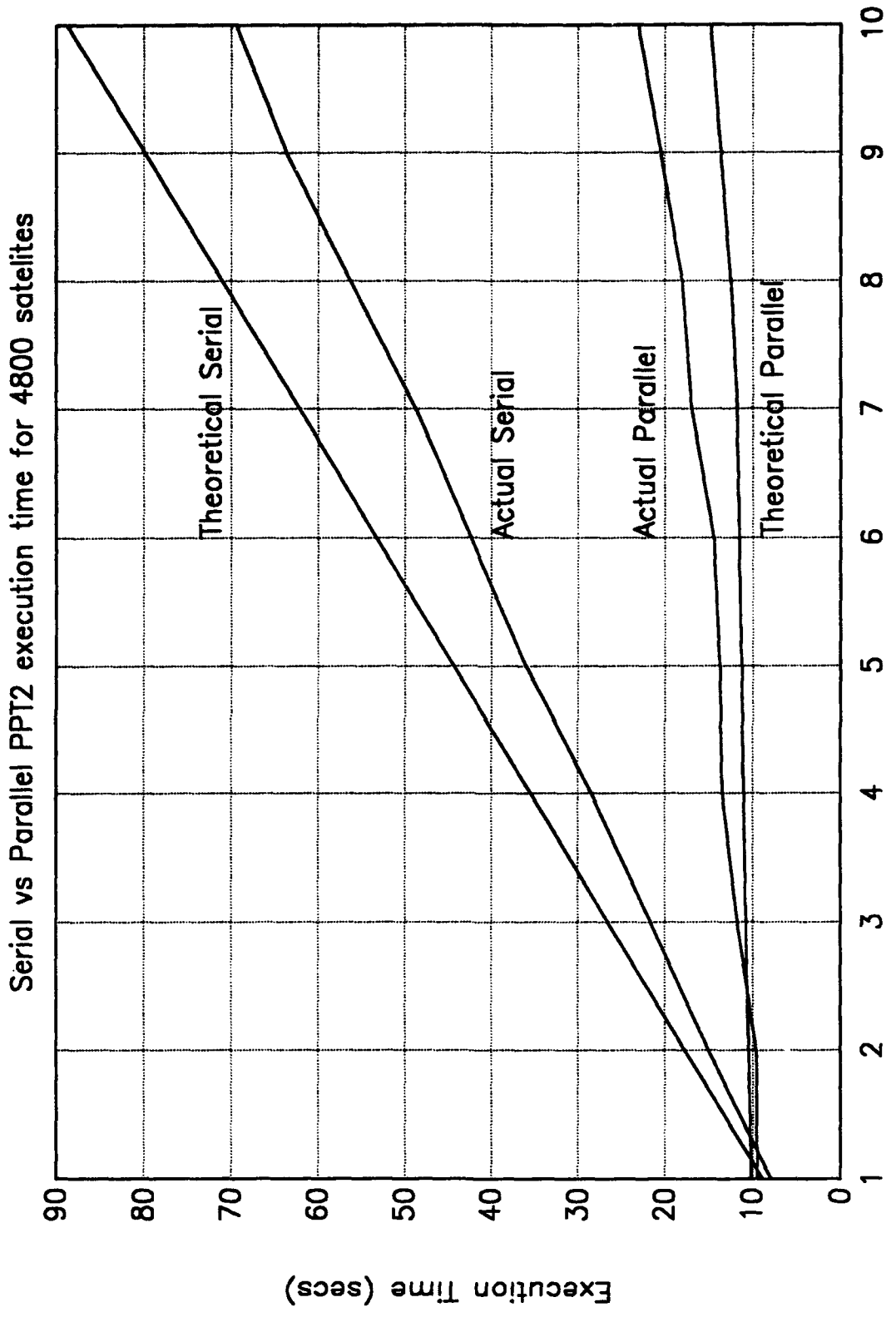


Figure 3.2. Serial vs. Parallel Results Using ds5 With Eight Workers.

Overall, the ds5 algorithm using PVM was able to process the satellite catalog faster than the serial program. These results were observed when the ECE network was being heavily used and also when the network had little activity on it. Also, the empirical data for the actual program times in Figure 3.2 is merely a representative result of executing the programs at one certain time of the day, and different numbers were obtained at different times, but again, the relative performance results were the same.

D. SPEEDUP COMPARISON

One standard figure of merit in comparing two algorithms is speedup. Speedup in this case, is the ratio of the serial results to the parallel results. The same data from the previous section was used to determine the speedup ratios for values of A ranging from one to ten. The speedup results are shown in Figure 3.3. Even though the actual speedup was less than expected, there was a definite decrease in execution time, thus an increase in speedup, when parallel execution was used instead of serial execution.

E. OPTIMUM NUMBER OF PROCESSORS TO USE

The execution time savings have been demonstrated in the previous sections, but one other question of interest is what the optimum number of processors to use would be. The optimum number of processors to use can be determined by setting the derivative of Equation 3.11, with respect to the variable k, equal to zero then solve for k. This will provide the optimum number of worker processors to use. Thus, by adding one processor for the supervisor node and one processor for the gathering node, the final value for the optimum number of processors is found.

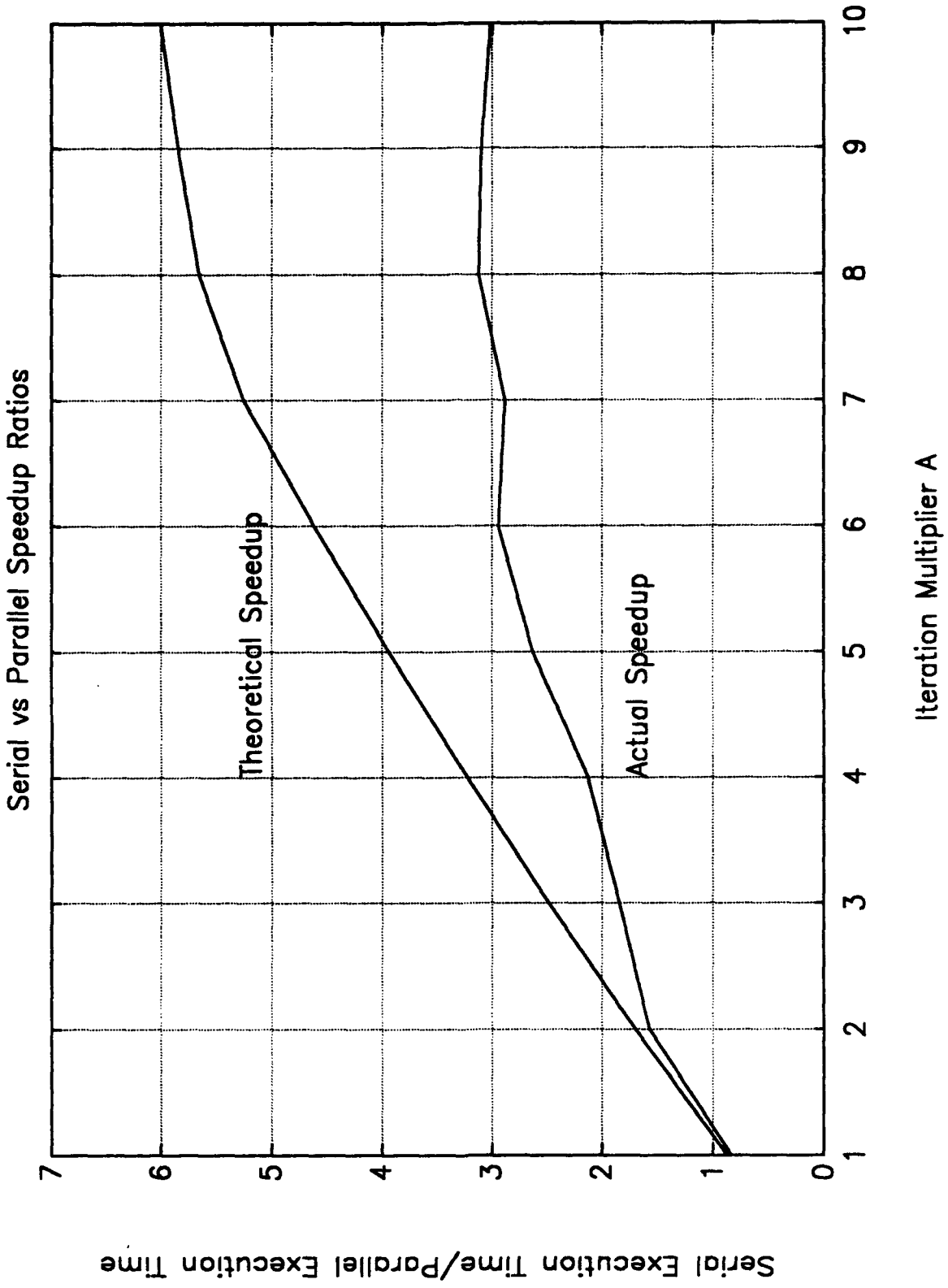


Figure 3.3. Serial vs. Parallel (ds5) Speedup Ratios Using Eight Workers.

Using Equation 3.7 the optimum product of worker processors and data blocks can be determined. When Equation 3.9 was used, only the number of worker processors can be found. The results for the two equations are given in Equations 3.12 and 3.13 respectively.

$$kn_b = \sqrt{\frac{S\left(C_{ps} + C_{upps} + A \cdot T_{ppt2}\right)}{C_f}} \tag{3.12}$$

$$k = \sqrt{\frac{S\left(C_{ps} + C_{upps} + A \cdot T_{ppt2}\right)}{C_f}} \tag{3.13}$$

With the exception of the number of blocks in Equation 3.12, both of these equations are identical. Equation 3.12 is much more flexible since the number of different processors available may be limited while the number of blocks is not. For example, using Equation 3.12, the empirical values in Appendix C, and setting $A = 2$, the optimum kn_b product is ≈ 63.35. If there are twelve total processors available, then by subtracting one processor for the gathering node and one processor for the supervisor node, there are ten processors available for the workers. Solving for the optimum number of blocks to send yields 6.335 resulting in n_b to be six or seven.

F. PPT2 AND PVM WITH ACTUAL DATA

As mentioned earlier, a sample of the satellite catalog was obtained. Though it was not used in determining which parallel algorithm to use or in ascertaining the values in Appendix C, it was used to produce plots similar to Figures 3.2 and 3.3. The data set contained 6795 satellite records. The serial vs. parallel comparison plot is provided in Figure 3.4, and the speedup comparison is shown in Figure 3.5. Again, the parallel algorithm ds5 was used with eight worker processors.

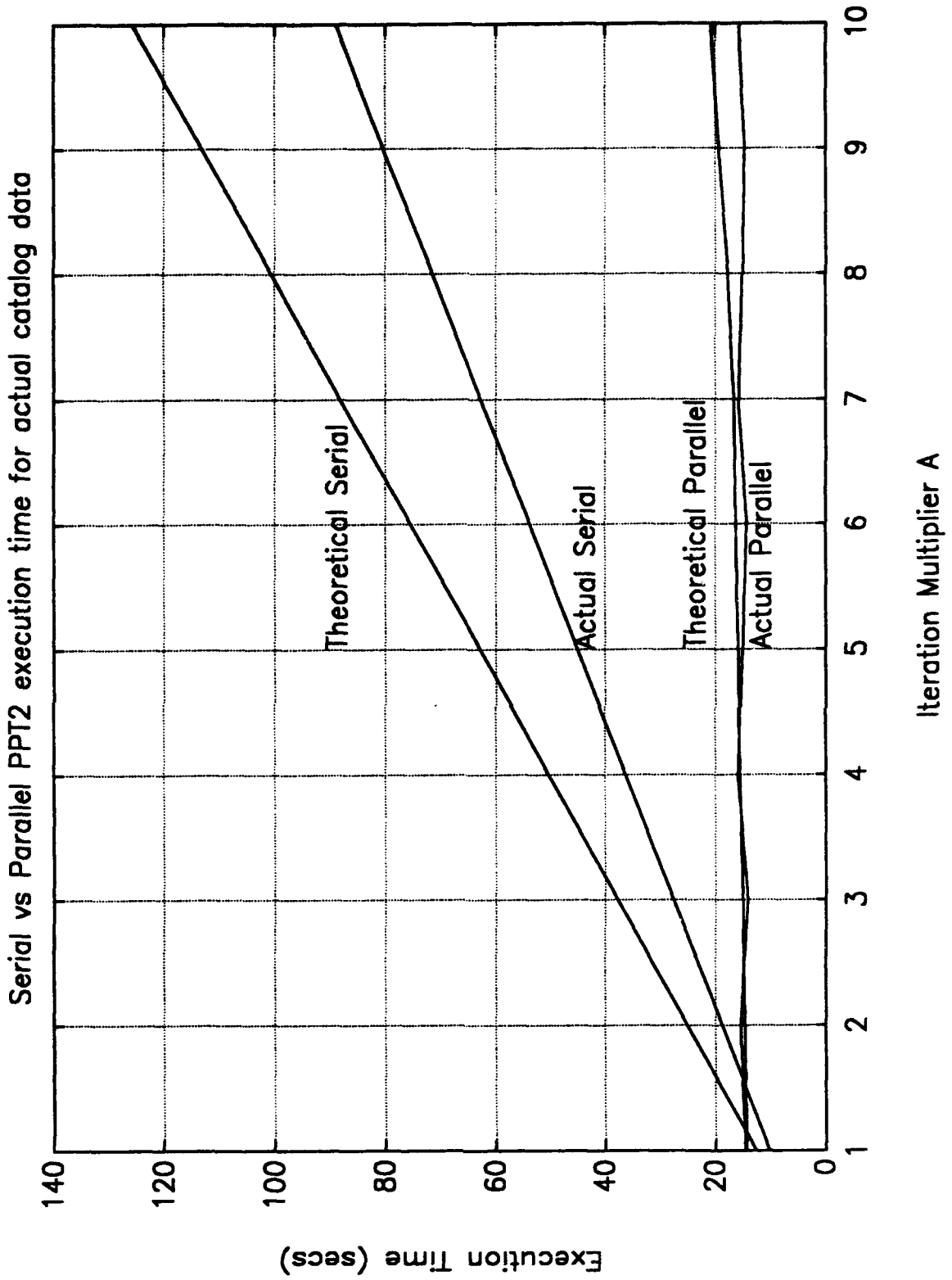


Figure 3.4. Serial vs. Parallel (ds5) Execution Time Comparisons Using Actual Data.

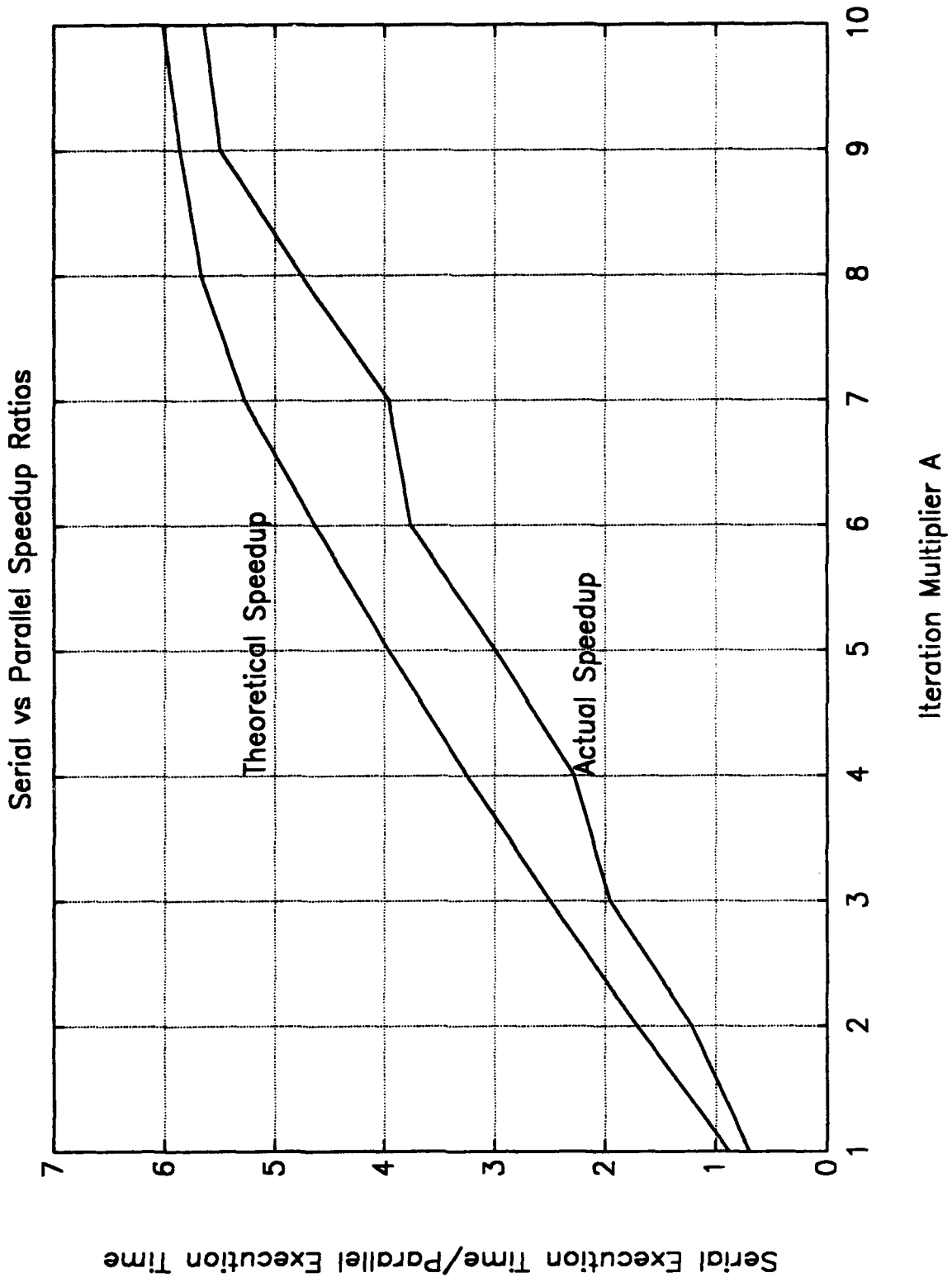


Figure 3.5. Serial vs. Parallel (ds5) Speedup Ratios Using the Catalog Data.

G. PPT2 CONCLUSIONS

The results of this chapter clearly demonstrate the effectiveness in reducing the overall execution time using a parallel algorithm. Also, this algorithm was run using a parallelization software tool, PVM, on a loosely coupled network of SUN workstations instead of a dedicated parallel multicomputer. Interestingly, the results for the actual catalog data were closer to the theoretical estimates than the data used in the previous sections. This validates the earlier results even though they were more conservative than the catalog results. Overall, using PVM and the multiblock data decomposition scheme resulted in improved PPT2 operation, which was the goal of this study.

IV. A THROUGHPUT CRITICAL ON-LINE APPLICATION

The second Naval application studied was the on-line hypothetical combat weapon system. The future combat systems will be large grain data flow, throughput-critical systems. These systems will be required to process electronic signals to detect, track, and determine a fire control solution for increasingly sophisticated threats. An example of the next generation of Navy signal processors is the AN/UYS-2 Digital Signal Processing System (also known as the Enhanced Modular Signal Processor, EMSP), which implements data-flow parallel processing to achieve high throughput rates for this type of environment in a single tightly coupled system [Ref. 4]. The hypothetical system presented here demonstrates the possible use of a loosely coupled LAN based cluster of processors in large-grain data-flow parallelization as against a tightly coupled system such as the EMSP.

A. PROCESSING IN A HYPOTHETICAL COMBAT SYSTEM

The hypothetical combat system is defined by the process node graph of Figure 4.1. This graph was designed to take into account the normal computational requirements of the combat system. The two left most branches, the paths through nodes P4D and P2B, represent the surface and air and the subsurface fire control solutions steps. The right most branch represents the surface and air tracking iterations. The nodes are marked with the processor it resides on and its personal identification letter. For instance, P4D stands for processor four, program D. The lines connecting the nodes have arrows indicating data flow paths. The numbers attached to the lines are a measurement of how large the data message is between the two nodes.

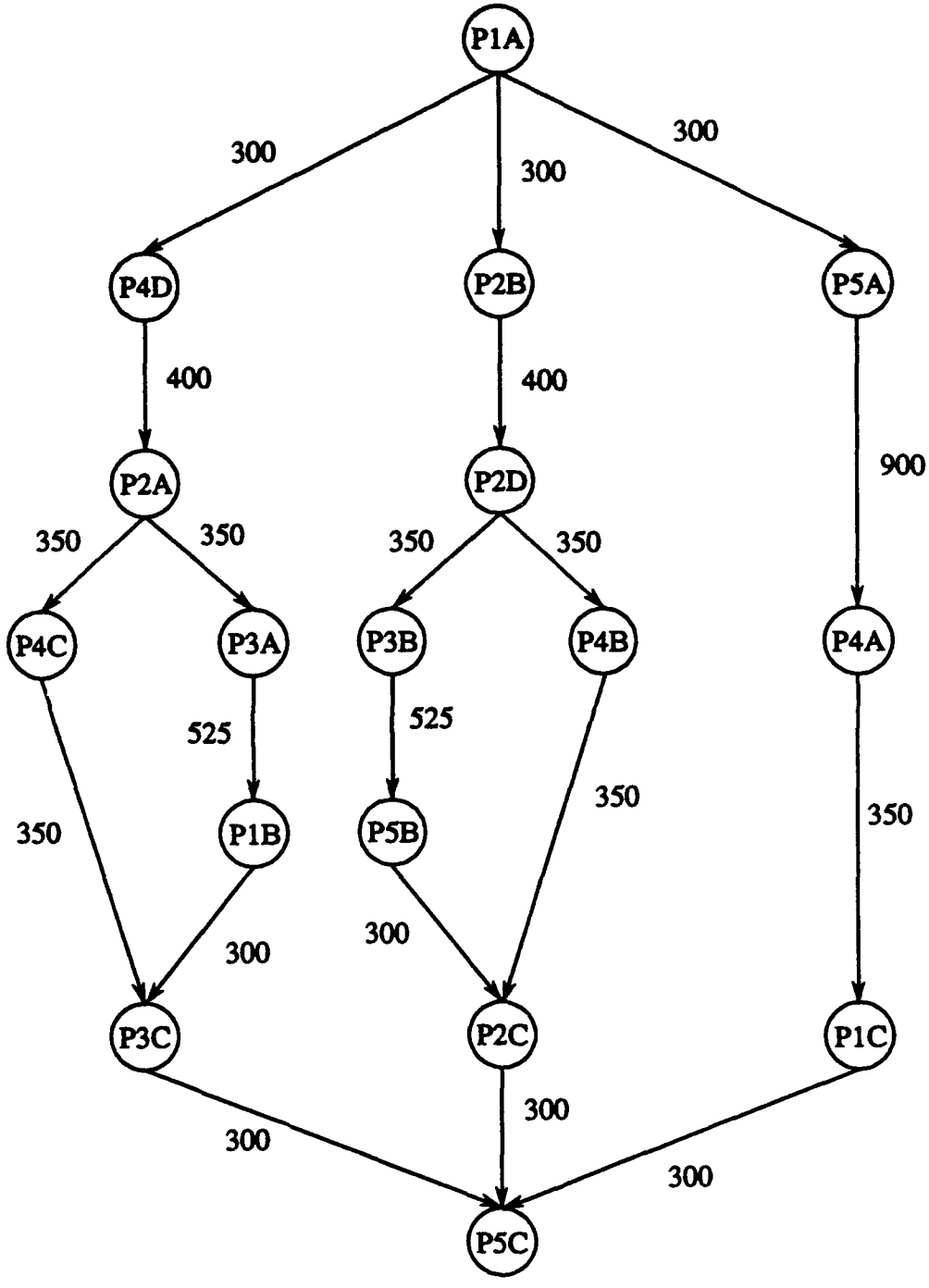


Figure 4.1. Hypothetical Combat System Node Representation Graph.

The cumulative communication and node execution time for each processor is approximately equal simulating a load-balanced graph. Another constraint given the graph is certain nodes must be allocated to certain processors due to memory dependencies. It is also assumed that all the nodes execute once in a given period which will be defined later. Though this is purely a hypothetical situation, it adequately simulates a possible on-line system.

B. PROBLEMS WITH IMPLEMENTATION USING PVM

PVM presented a few distinct problems for the on-line application. One problem is the high cost of buffer initialization associated with using PVM. Each PVM_INITSEND command, [Ref. 1], initializes a buffer in which to pack the output data. This cost is fixed and is independent of the amount of data to be sent. With many relatively small messages, this initialization time became an important factor in process execution time due to its additive affects.

Another problem occurred during program testing with added network loading. In Chapter V the loading will be discussed, but essentially a part of the forced network communications caused a slave program to send multiple, large messages to another. This sometimes caused the PVM daemon process on the slave's host computer to die. This occurrence has been reported before, [Ref. 5], but was not investigated because the use of the PVM_ADVISE command, [Ref. 1], eliminated this problem.

C. BATCHING OF COMMUNICATION COSTS

In PVM like systems, interprocessor communication has two distinct components, operating system (OS) related, and network related. The OS related part consumes processor cycles available to the application by making OS calls and affects the throughput. This could be regarded as OS contention between nodes on the same

processor. The network related part makes "available" processor cycles for one node, which is trying to transmit or receive data, unusable because other nodes have control of the bus. This is network contention and leads to processor blocking. Given the graph of Figure 4.1 with its multiple nodes on multiple processors these contentions can be numerous and affect the desired throughput.

One way of greatly reducing the number of these contentions is by batching the communication for each processor. Batching communication means what the name implies, taking all the input and output requirements for a processor and giving these tasks to one and only one node assigned to the processor. In order to accomplish this, it was assumed the nodes on a given processor could communicate using UNIX shared memory and that such communication was very cheap compared to PVM communication. This process added an extra node on each processor which is analogous to the gathering node described in Chapter II for the PPT2 algorithms.

The gathering node accesses the shared memory to gather the output data for transmission. It will also access the shared memory to place the input data upon reception. To do this, the shared memory is used in such a fashion that either the graph nodes can access their respective memory locations or the gathering node can access the entire memory, but not both.

D. THREE TECHNIQUES

The nodes were studied using three different methods of process execution. Of course, the overall graph execution was carried out in the sequence shown in Figure 4.1, but the sequence in which the nodes on each processor executed was manipulated. The three methods used a master/slave relationship. The master program took care of the PVM process spawning and then acted as either node P1A or Processor 1

depending on the technique chosen. The master would initiate an iteration then wait to receive certain criteria from the slaves before proceeding on to the next iteration. All the programs were written in C and are in the appendices as mentioned below. The three techniques are described as follows.

1. Unscheduled Node Processing

The unscheduled node processing method let each node begin execution upon receipt of data and communicate upon completion of execution. No attempt was made to reduce the number of contentions described in the communication batching section. In this technique, there is a PVM_SEND for every message. The results of this scheme was the metric by which the following "improvements" were judged. The code for this set of programs, one for each node, is in Appendix D.

2. Scheduled Node Processing

This method uses the scheduling method described in the last section of this chapter. In essence, all of the nodes on a given processor were restricted to a certain order in which they can execute thereby reducing the number of OS contentions. Shared memory use is assumed for communication between nodes on a processor. The batching of the communication between processors and the scheduling of nodes on the processors greatly reduces the network contentions. In this scheme, there is a PVM_SEND for every pair of communicating processors. The code for this technique is in Appendix E.

3. Scheduled Node Processing Using Hardware Multicasts

This technique uses basically the same approach as the previous method, but all communications are assumed to be passed between the nodes via hardware multicasts. Thus, all the communication from a processor to all the other processors is

multicast at the hardware level by the senders communication node. In this scheme, there is a PVM_SEND per processor. This PVM_SEND is assumed to be a hardware multicast to a group (which is not currently implemented in PVM). This further reduces the network contentions since fewer PVM message calls are used. Hardware multicasts were chosen because software multicasts greatly decreased the throughput. Using the PVM_MCAST command, [Ref. 1], was multicasting at the user level, but at the OS level, the PVM daemons were handling the multiple sends and receives. PVM routes messages either through the daemons or TCP direct. Since recent TCP implementations make use of hardware multicast for implementing user level multicasts, the use of hardware multicasts instead of the software commands was assumed. This is expected to be true of future PVM implementations. The code for this method is in Appendix F.

To further clarify the network contention reduction between the three algorithms, an example follows. From the graph in Figure 4.1, Processor 1 has three nodes. For the unscheduled method, Processor 1 has to output a total of five times per period. Using the scheduled technique, this number reduces to three. Then by using the hardware multicasts this number reduces to one. Of course, as the number of message pack and send calls is reduced the message size increases. This grouping of multiple messages reduces the number of times PVM has to initialize an output buffer eliminating this component of the communications cost overhead. However, in the last technique, every processor must unpack a larger message, reducing the gain from a hardware multicast.

E. NODE SCHEDULING

The last two processing techniques mentioned above depend on the nodes having a certain constraint on them as to when they can execute and communicate. To reduce the OS contentions each node is "scheduled" on its respective processor so each one has its turn without blocking another node or being blocked itself. Once the nodes are scheduled, it is instructive to think of their execution taking place within one frame of time slots. One of the assumptions, or constraints, applied to the hypothetical graph is the sum of the node execution and communication costs on each processor is approximately equal. This sum is the period in which one frame of scheduled time slots can be executed. To reduce the number of network contentions, the interprocessor communication is scheduled within each frame. Figure 4.2 shows a representative frame of time slots with the nodes from Figure 4.1 assigned to their respective execution positions.

P_1	P_2	P_3	P_4	P_5
com_i	D_{i-2}	B_{i-3}	B_{i-3}	A_{i-2}
A_i	com_i	C_{i-8}	C_{i-5}	B_{i-4}
Ai	A_{i-3}	com_i	D_{i-2}	C_{i-9}
B_{i-6}	B_{i-1}	A_{i-4}	com_i	C_{i-9}
C_{i-5}	C_{i-5}	B_{i-2}	A_{i-3}	com_i

Figure 4.2. Frame of Time Slots Starting at Time t_i.

Figure 4.2 shows the schedule of nodes for the ith frame. The node indices indicate which frame of data they are executing on in the current frame. For instance, P1A, the root node, is working on new data received for this frame, and P5C, the output node, is working on data the graph received i-9 frames ago. The schedule of

nodes is one of many possibilities, but once the schedule was chosen the indices used were unique.

To determine index j for node x_j, the following algorithm was applied.

Letting:

t_x = the time node x executes within the period.
t_{xp} = the time the parent of x executes within the period
t_{Pxc} = the time the processor x resides on communicates within the period.
t_{Pxpc} = the time the processor the parent of x resides on communicates within the period.
k = index of parent of x.

If x is the graph root node, then j = i.

If x and the parent of x reside on the same processor:
If $t_x < t_{xp}$ then j = k - 1.
If $t_x > t_{xp}$ then j = k.

If x and the parent of x reside on different processors:
If $t_{Pxc} < t_{Pxpc}$ then:
If $t_x > t_{Pxc}$ and $t_{xp} < t_{Pxpc}$ then j = k - 1.
If $t_x > t_{Pxc}$ and $t_{xp} > t_{Pxpc}$ then j = k - 2.
If $t_x < t_{Pxc}$ and $t_{xp} < t_{Pxpc}$ then j = k - 2.
If $t_x < t_{Pxc}$ and $t_{xp} > t_{Pxpc}$ then j = k - 3.

If $t_{Pxc} > t_{Pxpc}$ then:
If $t_x > t_{Pxc}$ and $t_{xp} < t_{Pxpc}$ then j = k.
If $t_x > t_{Pxc}$ and $t_{xp} > t_{Pxpc}$ then j = k - 1.
If $t_x < t_{Pxc}$ and $t_{xp} < t_{Pxpc}$ then j = k - 1.
If $t_x < t_{Pxc}$ and $t_{xp} > t_{Pxpc}$ then j = k - 2.

If a node relies on more than one parent, then use the above algorithm for all the parents then use the smallest calculated index out of the set of calculated indices for the node x.

The schedule represented by Figure 4.2 is not the only possible node scheduling scheme, but it was the one chosen for this study. Trying to determine an optimum schedule with respect to graph latency was not pursued.

V. RESULTS FOR THE ON-LINE APPLICATION

The three node processing techniques described in the previous chapter were implemented using PVM and certain parameters of performance were measured. The execution and communications costs were measured, as was done for PPT2, and theoretical values were obtained. The programs were run during a time when the network utilization was low and during a time when the network was purposefully loaded in order to compare and contrast the results.

A. PARAMETERS OF INTEREST

Throughput was the primary measurement studied. The values obtained were normalized with respect to the theoretical costs as discussed below. In addition to throughput, post processing of the data was used to determine the size an output buffer would need to be if there was a buffer between node P5C and the next stage of the weapons system. The buffer was accessed at the average period, $\bar{t}$. The standard deviation, s, of the period was determined to clarify the results of the buffer processing. For further statistical analysis, the coefficient of variation, V, which is defined as the ratio $s/\bar{t}$, was calculated. The scheduling represented by Figure 4.2 implies a graph latency of ten frames. Though this is a valid area of interest, output latency was not studied.

The theoretical period was determined by using the communications costs from Figure 4.1 and the execution times for the nodes in the longest path. The execution loop times and the message packing and sending times were measured on the ECE SUN system like the variables for PPT2 were determined. These numbers were used in combination with the variable weighting factor used when the programs were run to determine the theoretical period.

B. RESULTS WITHOUT NETWORK LOADING

The programs were run a multiple of times to determine what patterns, if any, they exhibited. Table 5.1 shows the results of one such run. For this run, the theoretical period was approximately 1.509 seconds. Even though empirical results are presented here, all the values were dependent on the load variations in the SUN network due to the other system users. While these values were obtained for this run, another set of runs at a different time could possibly yield different results. With this in mind, more emphasis was placed on the trends and patterns observed than on the actual values.

TABLE 5.1. TYPICAL ON-LINE RESULTS WITHOUT NETWORK LOADING.

Units = seconds	Unscheduled	Scheduled	Multicast
Average Period, $\bar{t}$	0.871	0.996	1.001
Normalized average, $\bar{t}_n$	57.7%	66.0%	66.3%
Standard deviation, s	0.339	0.0786	0.0729
Coefficient of variation, V	38.9%	7.89%	7.28%
Mean output buffer size, $\bar{b}$	3.56	1.431	1.896

From Table 5.1, the periods were slightly higher for the Scheduled and Multicast techniques than the Unscheduled method. This was the general trend for all the runs. Another trend was the fact the standard deviations of the Unscheduled method was between three to six times larger than the two scheduling algorithms. This was readily evident in the output which showed a wide range of throughput values for the

Unscheduled technique, and a more narrow range for the other two. The output buffer pattern observed was that the Unscheduled buffer size would be from two to five times larger than the other two.

The buffer size was observed to be more oscillatory for the Unscheduled processing than the other two approaches. Figures 5.1, 5.2, and 5.3 show the buffer size in reference to the iteration number for a run of 1000 graph iteration cycles utilizing the three node processing techniques. These plots reinforce the buffer data observations by showing the Unscheduled buffer size varying more, and getting larger than the Scheduled or Multicast methods.

C. RESULTS WITH A NETWORK PERTURBATION

The addition of a controlled load was applied to the process runs for this section. The loading consisted of one program manipulating large amounts of input/output, around 3.5 Mbytes, and two other programs sending and receiving a large amount of fairly large messages. These load programs were assigned to the same processors used by the graph nodes. The results from one of the runs are presented in Table 5.2.

This run was chosen because it presented some of the uncontrollable network influences as well as the observed trends. One example of the network usage affects is observed in the periods for the run prior to adding the load. The period for the Unscheduled method is noticeably less than the other two methods which is in contrast with the data in Table 5.1. This is due to the network load variations at the times the programs were executed.

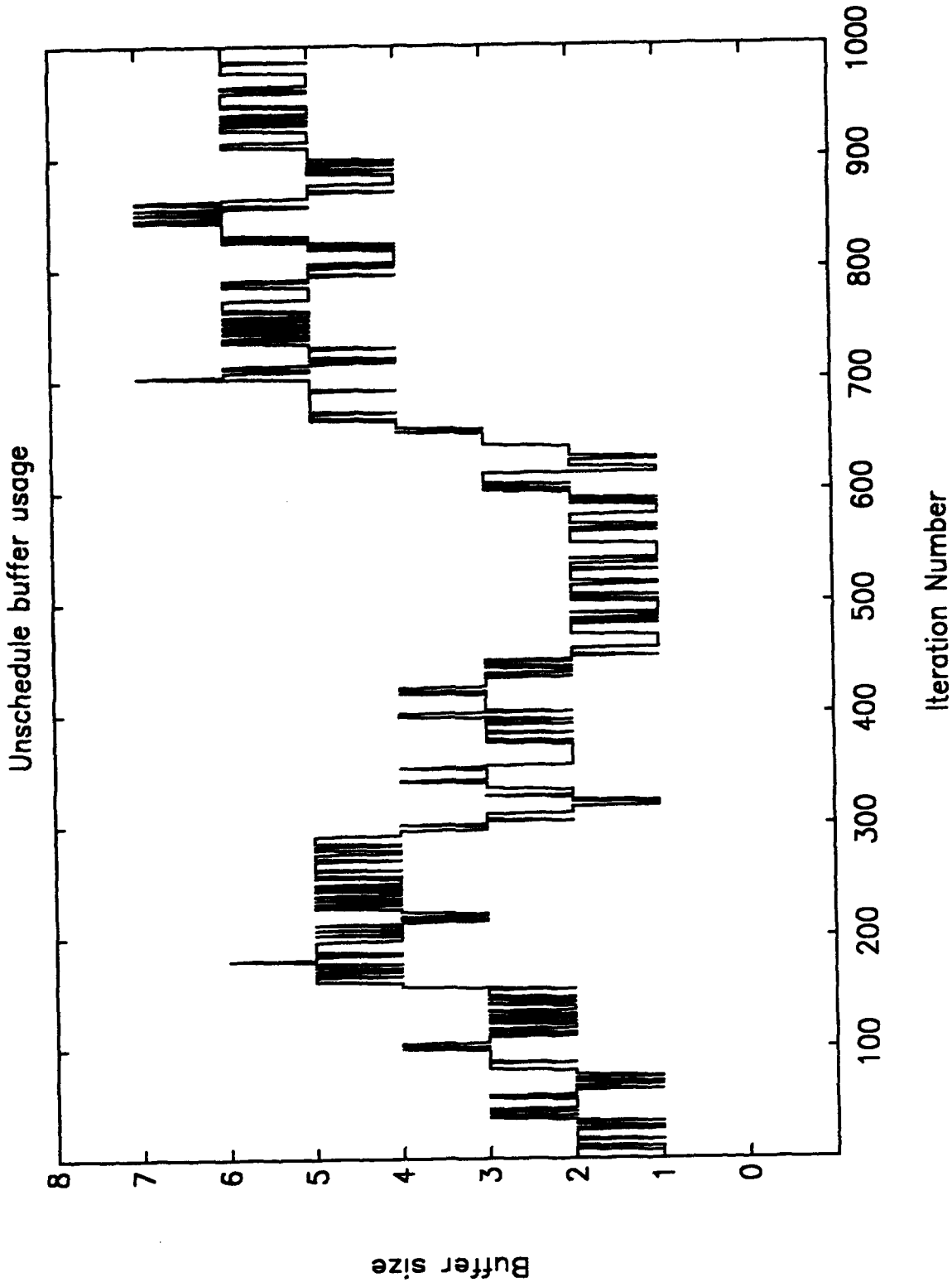


Figure 5.1. Unscheduled Output Buffer Size.

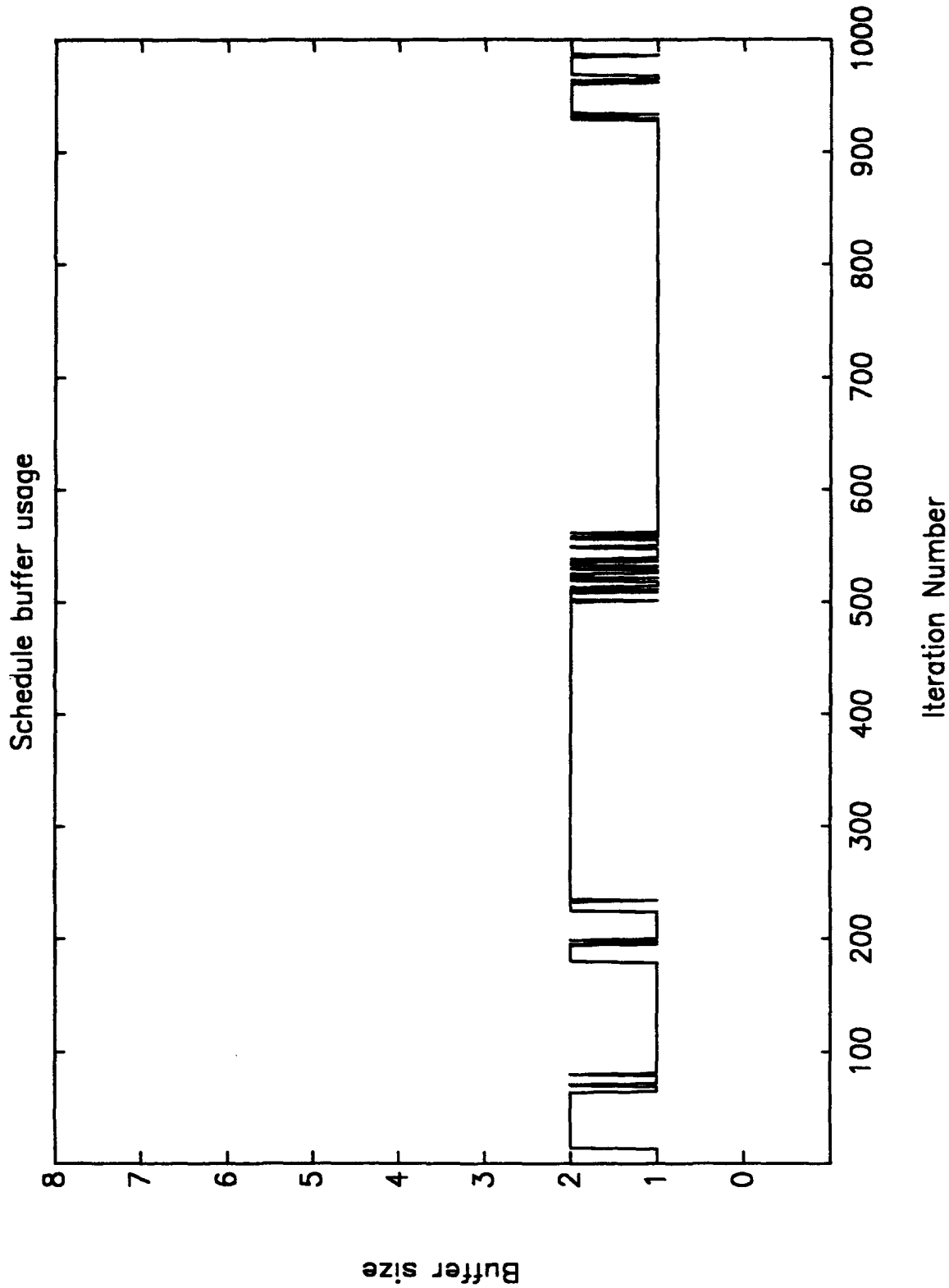


Figure 5-2. Scheduled Output Buffer Size.

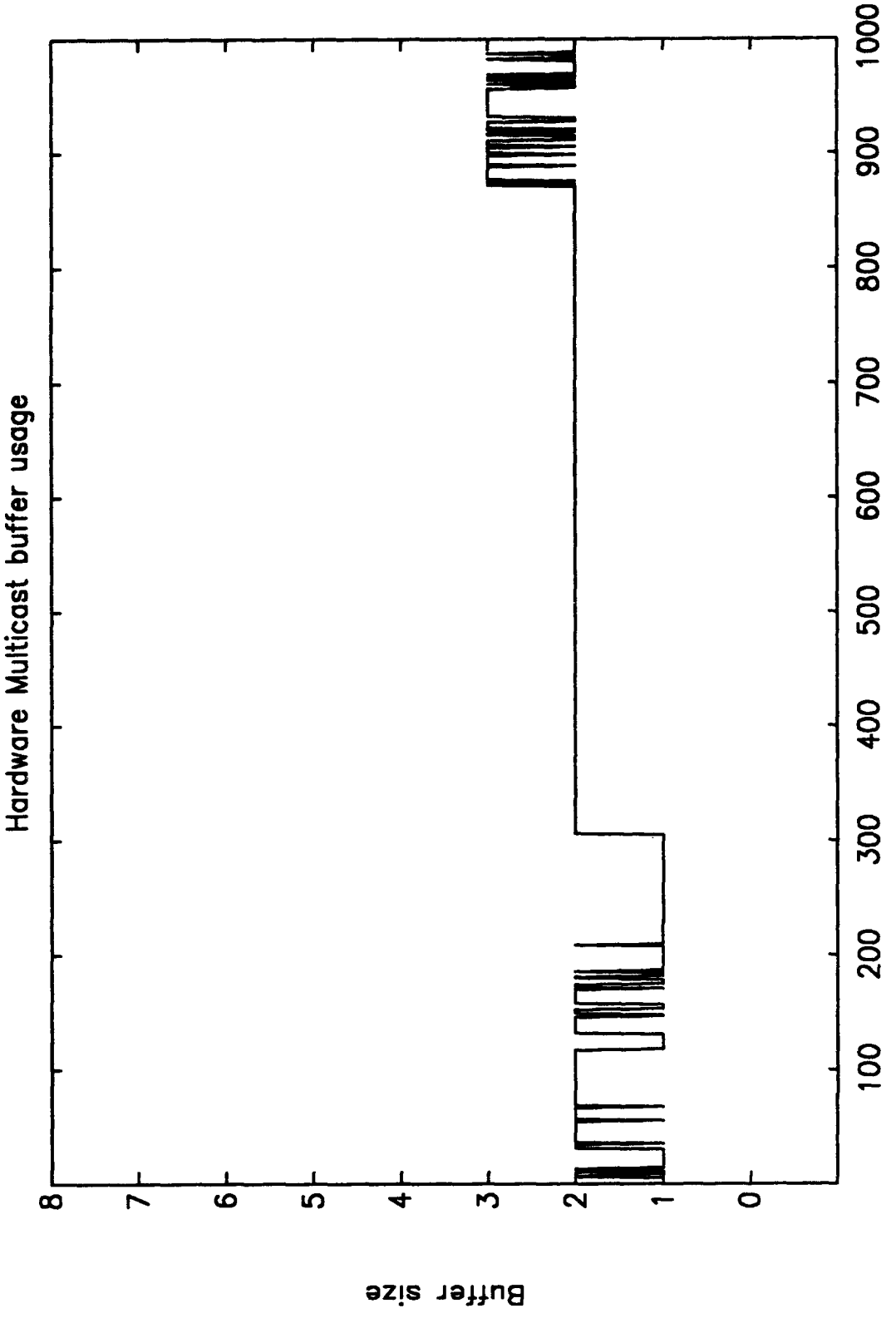


Figure 5-3. Hardware Multicast Output Buffer Size.

TABLE 5.2. TYPICAL ON-LINE RESULTS WITH NETWORK PERTURBATION.

Units = seconds	Unscheduled	Scheduled	Multicast
Before loading			
$\bar{t}$	0.784	0.996	1.072
$\bar{t}_n$	60.0%	66.0%	71.0%
s	0.256	0.0809	0.2489
V	32.7%	8.12%	23.2%
$\bar{b}$	1.78	1.587	2.14
During loading			
$\bar{t}$	1.094	1.611	1.566
$\bar{t}_n$	72.5%	106.8%	103.8%
s	0.277	0.1914	0.311
V	25.3%	11.88%	19.86%
$\bar{b}$	3.21	1.575	2.84

The observed patterns for each section of Table 5.2, before loading and during loading, were similar to those described in the previous section. The most prominent observation for this run comes from comparing the two sections. The buffer size stayed relatively constant before loading and during loading for the Scheduled and Multicast techniques, but the Unscheduled buffer size would increase by two to five times. The periods also increased, but not as significantly.

D. ON-LINE CONCLUSIONS

The use of node scheduling did not adversely affect the periods compared to not scheduling the nodes. While memory is cheap, and the buffer size may not be a hardware problem, the access time can be considerable compared to the throughput. This could add an excessive delay to the overall throughput of the graph when looking at it from the next stage after node P5C. This was the stimulus behind the buffer consideration.

The hardware multicasting did not considerably improve the performance of the node scheduling. This may be caused by the physical properties of the chosen graph because the nodes had to unpack one large message instead of a few smaller ones. The smaller messages were received at various times allowing the nodes to unpack them as the data arrived instead of all at one time. Another factor which influenced the Multicast performance was the fact the physical hardware was not available and PVM was used to simulate it.

VI. CONCLUSION

This thesis provided separate conclusions at the end of each application section. Overall, using a software tool can effectively improve the performance of a given procedure. PVM is becoming the standard for use as a parallelization software tool and it demonstrated its usefulness when applied to the off-line PPT2 program and the on-line hypothetical combat weapons system.

A. FUTURE STUDY

Further work is required in the following areas:

1. The PPT2 theoretical optimum block size could be studied further.

2. A larger set of data from the Naval Space Command would increase the usefulness of the results presented. If possible, the use of the block distribution algorithm and PVM on the actual satellite catalog with the proper number of iterations for each individual record would better demonstrate a real scenario.

3. The four data decomposition schemes presented for PPT2 are basic with numerous possible improvements. One such variation is having the supervisor send an initial block of data to each worker, divide up the remaining records into blocks, then send these blocks. Another area for testing is the use of multiple supervisors with their own sets of workers implementing each of the schemes.

4. The way in which the network was loaded for the on-line application was not varied. The load programs ran on the same processors the node programs were on. Further study on the affects of the load programs operating on different processors is warranted.

5. The on-line application research just scratched the surface of the possibilities for this area. The code for the node processing schemes were written with the user able to

easily vary the communications and execution costs. Though many program runs were accomplished for this thesis, the varying of the costs was not fully studied.

LIST OF REFERENCES

1 Oak Ridge National Laboratory, Oak Ridge, Tennessee Technical Report ORNL/TM-12187, *PVM 3 User's Guide and Reference Manual*, by A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, and V. Sunderam, May, 1993.

2 Phipps, W. E., *Parallelization of the Navy Space Surveillance Center (NAVSPASUR) Satellite Motion Model*, Master's Thesis, Naval Postgraduate School, Monterey, California, June, 1992.

3 Phipps, W.E., Neta, B., and Danielson, D.A.,"Parallelization of the Naval Space Surveillance Satellite Motion Model," *Journal of Astronautical Sciences*, Vol. 42, No. 2, April-June 1993.

4 Rice, M.L., "The Navy's new standard digital signal processor: The AN/UYS-2," paper presented at the Association of Scientists and Engineers, 27th Annual Technical Symposium, 23 May 1990.

5 Lewis, M. J., and Cline, R. E. Jr., "PVM Communication Performance in a Switched FDDI Heterogeneous Distributed Computing Environment," paper presented at the Symposium on Reliable and Distributed Systems, October 1993.

APPENDIX A - ACQUIRING AND INSTALLING PVM

First send the message "send index from pvm3" to netlib@ornl.gov
then follow the instructions.

Probably the quickest way to get the files is to use rcp.
finger anon@netlib2.cs.utk.edu
(* this command will explain how to copy files from the Netlib Software Repository *)

It will tell you to use: rcp anon@netlib2.cs.utk.edu:FILENAME LOCAL_FILENAME

Create the directory "pvm" where pvm is to be installed.

So type the following commands from the pvm directory:

rcp anon@netlib2.cs.utk.edu:pvm3/filename .
or
rcp -r anon@netlib2.cs.utk.edu:pvm3/directory .

for all the files listed in the index.

At some point in time, the access modes for all files should be changed to allow all users to be able to read and execute them.

Next, type: sh pvm3.1.shar
This command will create the pvm3 subdirectory and extract the pvm files.

more pvm/pvm3/lib/cshrc/stub
This command shows a portion of code that needs to be appended to the installers .cshrc file. The "setenv PVM_ROOT " line must be modified to take into account the current location of pvm.

pvm/pvm3/make all
This command will then compile the pvm source code. Look in the file Makefile for individual options if you do not want to install everything. If errors occur, the Makefile.body needs to be modified then ../lib/UpdateMk needs to be run. For instance, in the file Makefile in the xep subdirectory, the xcflags path had to be changed in order to get xep installed (only occurred when installing 3.1, had no problems installing 3.2.

APPENDIX B - BLOCK DECOMPOSITION SCHEME PROGRAMS

SUPERVISOR (MASTER)PROGRAM:

```
      program thesis1m
      include 'fpvm3.h'
c -------------------------------------------------------
c Fortran Master program to solve the NAVASPASUR satellite
c orbit prediction problem.
c This program reads the data from a file and distributes the data to
c the working nodes one block at a time.
c -------------------------------------------------------

      implicit real*8 (a-h,o-z)
      character*16 filenm

      integer pid, bsz
      integer eof, gattime(60)
      integer start, finish, endtime, Gettime
      external Gettime !$pragma C( gettime )

      common/bloc/sat(84,8000)

      data istop/1/,pid/0/,msglen/672/
      data isat/1/,n/1/

      integer i, info, nproc, iter
      integer mytid, tids(0:40), slvtime(16)
      integer who
      character*12 nodename
      character*8 arch

*     Enroll this program in PVM
      call pvmfmytid( mytid )

c ------------ Starting up all the tasks ----------------------

*     Initiate nproc instances of  thesis1s slave program
      print *,'How many working slave programs (1-16)?'
      read *, nproc
      nproc=nproc+1
      print*,' '
      print*, 'Which input file?'
      read*, filenm
      print*, 'What blocksize?'
      read*, bsz
      print*, 'How many iterations?'
      read*,  anum

*------Read complete catalog of satellite data----------
```

```
      open(10,file=filenm)
      read(10,*,iostat=eof)(sat(j,1),j=1,84)

      isat=0

200   if(eof.ge.0)then
        isat=isat+1
        read(10,*,iostat=eof)(sat(j,isat+1),j=1,84)
        go to 200
      endif

      close(10)

c     Print the number of satellite records received.

      print*,'isat = ',isat
      print*,'     '

*     If arch is set to '*' then ANY configured machine is acceptable
*     otherwise arch should be set to architecture type you wish to use.
      nodename = 'thesis1s'
      arch = '*'

      call pvmfspawn( nodename, 0, arch, nproc, tids, info )



c ----------** Begin user program **----------------

c ---------------- Get beginning time stamp ----------------

      start = Gettime ( start )


*     send number of satellites and slave id array to slaves

      msgtype = 2
      call pvmfinitsend(0, info)
      call pvmfpack( INTEGER4, nproc, 1, 1, info )
      call pvmfpack( INTEGER4, tids, nproc, 1, info )
      call pvmfpack( INTEGER4, isat, 1, 1, info )
      call pvmfpack( INTEGER4, bsz, 1, 1, info )
      call pvmfpack( INTEGER4, anum, 1, 1, info )
      call pvmfmcast( nproc, tids, msgtype, info)


*     broadcast data to all node programs
```

```
c     Determine how many records in each block
      iter=isat/((nproc-1)*bsz)
      iterx=mod(isat/bsz,nproc-1)

      do 305 j=1,bsz

      do 300 i=1,nproc-1

        if(i .le. iterx) then
          itern = iter + 1
        else
          itern = iter
        endif

        call pvmfinitsend( 0, info )
        call pvmfpack( BYTE1,  sat(1,n), msglen*itern, 1, info )
        call pvmfsend( tids(i), 3, info )

300     n=n+itern

305     continue

c     wait for data completion signal from gathering node
      call pvmfrecv( tids(0), 4, info )
      call pvmfunpack( INTEGER4, gattime, 33, 1, info )


c ---------- Collect ending time stamp ----------

      finish = Gettime ( finish )
      endtime = finish - start

      print*, 'The end to end runtime is ',endtime, ' usecs.'


c ---------- End user program ----------

c     program finished leave PVM before exiting
      call pvmfexit(info)
      stop
      end
```

WORKER (SLAVE) PROGRAM:

```
      program thesis1s
      include 'fpvm3.h'

c ------------------------------------------------------
c Fortran Slave program to solve the NAVASPASUR satellite
c orbit prediction problem.  The slave program consists of two nodes.
c One node is a position calculation node and the other is the data
c colection node.  This version of the master-slave configuration
c receives blocks of data from the master and performs calculations.
c ------------------------------------------------------

      implicit real*8 (a-h,o-z)
      real*8 kf(10)

      integer pid, me, nproc, bsz, gattime(60)
      integer start, finish, endtime, Gettime
      external Gettime !$pragma C( gettime )

      common/cons/a(64)
      common/ppt/f(25),osc(10),kf(10),cf(10),bs(3,4),u(3),v(3),w(3),r,
     &   vel(3),dind,tm,dkz,dident
      common/dcsub/pe(6,8),e(8,8),ep(8,8),g(8),gp(8),ifti(8),ifto(8),
     &   iteri,itero,jof,jol,stat(20),tol(6),iw,of(11),ow(8,8)
      common/foreo/rho(3),ros,hdr,hdv,rdv,del,iter
      common/bloc/sat(84,8000)

      data istop/1/,pid/0/,msglen/672/
      data isat/1/,n/1/

      integer  info, mytid, mtid, msgtype
      integer  tids(0:40)

c---- Enroll this program in PVM -----
      call pvmfmytid( mytid )

c     Get the master's task id
      call pvmfparent( mtid )


c ------** Begin user program **----------

c     Receive data from host
      call pvmfrecv( mtid, 2, info )
      call pvmfunpack( INTEGER4, nproc, 1, 1, info )
      call pvmfunpack( INTEGER4, tids, nproc, 1, info )
      call pvmfunpack( INTEGER4, isat, 1, 1, info )
      call pvmfunpack( INTEGER4, bsz, 1, 1, info )
      call pvmfunpack( INTEGER4, anum, 1, 1, info )
```

```
c-------------------------------------------------------
c  Determine which slave I am
   do 10 i=0, nproc-1
      if( tids(i) .eq. mytid ) me = i
 10   continue


c -------------------------------------------------------
c     Determine if I am the gathering slave

      if( tids(0) .eq. mytid ) then

c     Execute the gathering node
*     Begin Collecting Node

      msgtype = 10
      k=1

      do 1000, i=1,(nproc-1)*bsz
       call pvmfrecv( -1, msgtype, info)
       call pvmfunpack(INTEGER4, iter, 1, 1, info)
       call pvmfunpack(BYTE1, sat(1,k), msglen*iter, 1, info)
       k=(i)*iter+1
1000   continue

c      Commented out since I/O time was not considered
*      Write results to external file

*       open(6,file='/home3/stone/pvm3/bin/SUN4/thesis1.out')

*       do 1231 i=1,isat
* 1231     write(6,*)(sat(j,i),j=1,84)

*       close(6)


*  Send message to Host that process is complete
       msgtype = 4
       call pvmfinitsend(  0, info)
       call pvmfpack( INTEGER4, gattime, 33, 1, info )
       call pvmfsend( mtid, msgtype, info )

*  End Collecting Node


c-----Begin Working Nodes----------------------
```

```
      else

*     Determine my block size
      iter=isat/((nproc-1)*bsz)
      iterx=mod(isat/bsz,nproc-1)


c ---------------  beginning time stamp ------------

      start = Gettime ( start )

      call cons1
      msgtype = 3

      do 1405 j=1,bsz

      if(me .le. iterx)then
         itern=iter+1
        else
         itern=iter
        endif

       k=(j-1)*iter+1


       call pvmfrecv( mtid, msgtype, info )
       call pvmfunpack( BYTE1, sat(1,k), msglen*itern, 1, info )

      do 1400 i=1,itern

c-----Receive satellite to process ----


       do 1380 n=1,84
1380      f(n)=sat(n,k+i-1)

*     Set parameters for subroutine ppt2

        ind=1
        kz=idint(dkz)

*     Compute secular recovery

        call ppt2(ind,kz)

*     Compute subsequent task, ie. predict position, update elements ...

        ind=idint(dind)
        call ppt2(ind,kz)
```

```
      do 1390 n=1,84
1390     sat(n,k+i-1)=f(n)

1400  continue


c --------Send computed results to gathering node----------
      call pvmfinitsend( 0, info )
      call pvmfpack( INTEGER4, itern, 1, 1, info)
      call pvmfpack( BYTE1, sat(1,k), msglen*itern, 1, info )
      call pvmfsend( tids(0), 10, info )

1405  continue

c ---------------  Ending time stamp ---------------

    finish = Gettime ( finish )

    endtime = finish - start

c    The following was used for trouble shooting and processor comparison
*    call pvmfadvise(PvmRouteDirect,info)

*    msgtype = 25
*    call pvmfinitsend( 0, info )
*    call pvmfpack( INTEGER4, endtime, 1, 1, info )
*    call pvmfsend( mtid, msgtype, info)

    end if

c -------- End user program -------

c    Program finished. Leave PVM before exiting
    call pvmfexit(info)
    stop
    end

**********************************************************************
*DECK PPT2

*****     NAVAL SPACE COMMAND PROPRIETARY CODE SEE      *****
*****     PROFESSOR B. NETA, NAVAL POSTGRADUATE         *****
*****     SCHOOL FOR ACCESS TO THE PPT2 SOURCE CODE     *****
```

APPENDIX C - EMPIRICAL VALUES FOR PPT2 VARIABLES

The ECE network results presented here were using SUN/SPARC IPX and SUN/SPARC II stations. Esentially no difference in output values were observed between the two stations.

Variable	Definition	Value
S	total number of satellites in the input file	4800
t_o	node process initialization time	5506.7 μs
t_{gm}	time for gathering node to report to the supervisor the process is complete	1300 μs
n_b	number of blocks sent to each worker	4
C_f	fixed communications time for buffer setup and network access for sending records	6027.84 μs
C_{ps}	communications time required to pack and send one satellite record	1264.52 μs
C_{upf}	fixed communications time to unpack the input buffer	132.98 μs
C_{upps}	communications time to unpack one satellite record	75.7 μs
k	number of working processors used	8
S_p	number of satellites sent to each worker = S/k	600
S_b	number of satellites per data block = S_p/n_b	150
T_{ppt2}	time for PPT2 to operate on one satellite record	1850 μs

APPENDIX D - UNSCHEDULED NODE PROCESSING PROGRAMS

MASTER PROGRAM (NODE P1A):

```
/*****************************************************************************
Unscheduled master program, also handles node P1A communication and execution
requirements.
*****************************************************************************/
#include "pvm3.h"
#include <stdio.h>
#include <sys/time.h>
#include <time.h>
#include <sys/types.h>
#include <signal.h>
#include <stdlib.h>

    /* CONSTANTS */
#define done_lp  1000          /* Loop iteration counter */
#define snl      400           /* Iteration number for start of network loading */
#define dosize   300           /* Size of noise message */

    /* GLOBAL VARIABLES */
int done = 0;
int who;
int pnum;
double data_mat[5000];
double do_again;
int ld_num = 55000;

    /* SLAVE VARIABLES */
int  comm_gain ;       /* For varying the communication weights */
int  my_wt;            /* For varying the execution weights */
int  in4df1a = 300;    /* The next variables contain the branch communication */
int  in2bf1a = 300;    /*  and are defined as, using in4df1a, input to node */
int  in5af1a = 300;    /*  P4D from P1A */
int  in2af4d = 400;
int  in2df2b = 400;
int  in4af5a = 900;
int  in4cf2a = 350;
int  in3af2a = 350;
int  in3bf2d = 350;
int  in4bf2d = 350;
int  in1bf3a = 525;
int  in5bf3b = 525;
int  in3cf4c = 350;
int  in3cf1b = 300;
int  in2cf5b = 300;
int  in2cf4b = 350;
int  in1cf4a = 350;
int  in5cf1c = 300;
```

```
int  in5cf2c = 300;
int  in5cf3c = 300;

main()
{
  char SLAVENAME[3];
  int i, k;
  int rnum, rnumtot;
  int n_tm_max = 0, wnl = 0;
  int nl_done = 0;
  char myname[5];
  int nproc = 16;
  int mytid;                                  /* my task id */
  int tids[20], stids[5];              /* slave task ids */
  int snproc = 3;
  struct itimerval tmrval;

/* read in communication and execution scale factors */
  printf("\nComm wt =  ");
  scanf("%d", &comm_gain);
  printf("\Ex wt =  ");
  scanf("%d", &my_wt);
  my_wt = my_wt*4;

/* use loading or not */
  printf("\With NET loading type 1, without NET loading type 2:  ");
  scanf("%d", &wnl);


/* initialize matices */
  for(k=0; k<1500; k++)
    data_mat[k]=(double)k+5.66666;

/* enroll in pvm */
  mytid = pvm_mytid();

/* start up slave tasks */
  nproc = 16;

  gethostname(myname,5);

  pvm_spawn("p1b", NULL, 1, myname, 1, &tids[0]);
  pvm_spawn("p1c", NULL, 1, myname, 1, &tids[1]);
  pvm_spawn("p2a", NULL, 1, "sun3", 1, &tids[2]);
  pvm_spawn("p2b", NULL, 1, "sun3", 1, &tids[3]);
  pvm_spawn("p2c", NULL, 1, "sun3", 1, &tids[4]);
  pvm_spawn("p2d", NULL, 1, "sun3", 1, &tids[5]);
  pvm_spawn("p3a", NULL, 1, "sun8", 1, &tids[6]);
  pvm_spawn("p3b", NULL, 1, "sun8", 1, &tids[7]);
  pvm_spawn("p3c", NULL, 1, "sun8", 1, &tids[8]);
  pvm_spawn("p4a", NULL, 1, "sun9", 1, &tids[9]);
```

```
    pvm_spawn("p4b", NULL, 1, "sun9", 1, &tids[10]);
    pvm_spawn("p4c", NULL, 1, "sun9", 1, &tids[11]);
    pvm_spawn("p4d", NULL, 1, "sun9", 1, &tids[12]);
    pvm_spawn("p5a", NULL, 1, "sun20", 1, &tids[13]);
    pvm_spawn("p5b", NULL, 1, "sun20", 1, &tids[14]);
    pvm_spawn("p5c", NULL, 1, "sun20", 1, &tids[15]);

/* Send initial book keeping data to all the slaves */
    pvm_initsend(PvmDataDefault);
    pvm_pkint(&nproc, 1, 1);
    pvm_pkint(tids, nproc, 1);
    pvm_pkint(&my_wt, 1, 1);
    pvm_pkint(&comm_gain, 1, 1);
    pvm_mcast(tids, nproc, 20);

/* Send the input and output costs for each slave */
    pvm_initsend(PvmDataDefault);
    pvm_pkint(&in1bf3a, 1, 1);
    pvm_pkint(&in3cf1b, 1, 1);
    pvm_send(tids[0], 25);

    pvm_initsend(PvmDataDefault);
    pvm_pkint(&in1cf4a, 1, 1);
    pvm_pkint(&in5cf1c, 1, 1);
    pvm_send(tids[1], 25);

    pvm_initsend(PvmDataDefault);
    pvm_pkint(&in2af4d, 1, 1);
    pvm_pkint(&in4cf2a, 1, 1);
    pvm_send(tids[2], 25);

    pvm_initsend(PvmDataDefault);
    pvm_pkint(&in2bf1a, 1, 1);
    pvm_pkint(&in2df2b, 1, 1);
    pvm_send(tids[3], 25);

    pvm_initsend(PvmDataDefault);
    pvm_pkint(&in2cf4b, 1, 1);
    pvm_pkint(&in2cf5b, 1, 1);
    pvm_pkint(&in5cf2c, 1, 1);
    pvm_send(tids[4], 25);

    pvm_initsend(PvmDataDefault);
    pvm_pkint(&in2df2b, 1, 1);
    pvm_pkint(&in3bf2d, 1, 1);
    pvm_send(tids[5], 25);

    pvm_initsend(PvmDataDefault);
    pvm_pkint(&in3af2a, 1, 1);
    pvm_pkint(&in1bf3a, 1, 1);
    pvm_send(tids[6], 25);
```

```
pvm_initsend(PvmDataDefault);
pvm_pkint(&in3bf2d, 1, 1);
pvm_pkint(&in5bf3b, 1, 1);
pvm_send(tids[7], 25);

pvm_initsend(PvmDataDefault);
pvm_pkint(&in3cf4c, 1, 1);
pvm_pkint(&in3cf1b, 1, 1);
pvm_pkint(&in5cf3c, 1, 1);
pvm_send(tids[8], 25);

pvm_initsend(PvmDataDefault);
pvm_pkint(&in4af5a, 1, 1);
pvm_pkint(&in1cf4a, 1, 1);
pvm_send(tids[9], 25);

pvm_initsend(PvmDataDefault);
pvm_pkint(&in4bf2d, 1, 1);
pvm_pkint(&in2cf4b, 1, 1);
pvm_send(tids[10], 25);

pvm_initsend(PvmDataDefault);
pvm_pkint(&in4cf2a, 1, 1);
pvm_pkint(&in3cf4c, 1, 1);
pvm_send(tids[11], 25);

pvm_initsend(PvmDataDefault);
pvm_pkint(&in4df1a, 1, 1);
pvm_pkint(&in2af4d, 1, 1);
pvm_send(tids[12], 25);

pvm_initsend(PvmDataDefault);
pvm_pkint(&in5af1a, 1, 1);
pvm_pkint(&in4af5a, 1, 1);
pvm_send(tids[13], 25);

pvm_initsend(PvmDataDefault);
pvm_pkint(&in5bf3b, 1, 1);
pvm_pkint(&in2cf5b, 1, 1);
pvm_send(tids[14], 25);

pvm_initsend(PvmDataDefault);
pvm_pkint(&in5cf1c, 1, 1);
pvm_pkint(&in5cf2c, 1, 1);
pvm_pkint(&in5cf3c, 1, 1);
pvm_send(tids[15], 25);

/* If want loading */
  if ( wnl == 1)
  {
```

```
    pvm_spawn("s1", NULL, 1, "sun3", 1, &stids[0]);
    pvm_spawn("s2", NULL, 1, "sun20", 1, &stids[1]);
    pvm_spawn("s3", NULL, 1, "sun8", 1, &stids[2]);
    .
    pvm_initsend(PvmDataDefault);
    pvm_pkint(&snproc, 1, 1);
    pvm_pkint(stids, snproc, 1);
    pvm_mcast(stids, snproc, 62);
  }

/* Begin User Program */

  for (done = 0; done < done_lp; done ++)
  {

    if ( done == snl && wnl == 1)
    {
      pvm_initsend(PvmDataDefault);
      pvm_pkint(&ld_num, 1, 1);
      pvm_send(stids[0], 22);

    }

    if (done == done_lp - 1)
      data_mat[0] = -444.555;

    pvm_initsend(PvmDataDefault );
    pvm_pkdouble(data_mat, dosize*comm_gain, 1);
    pvm_send(tids[3], 4);

    pvm_initsend(PvmDataDefault );
    pvm_pkdouble(data_mat, dosize*comm_gain, 1);
    pvm_send(tids[12], 13);

    pvm_initsend(PvmDataDefault );
    pvm_pkdouble(data_mat, dosize*comm_gain, 1);
    pvm_send(tids[13], 14);

    for (k = 0; k < my_wt*2; k++)
      for (i = 0; i < 1360; i++)
      {
        rnum = rand();
        rnumtot = rnumtot + rnum;
        data_mat[i+1] = i + 1;
      }

    printf("\n On loop number %d\n",done);

      pvm_recv(tids[1], 25);
      pvm_upkdouble(&do_again, 1, 1);
```

```
  }    /* end of for loop */

  printf("\nThe loop is done\n");


/* Ensure all slaves have quit prior to termination */
  pvm_recv(tids[15], 35);
  pvm_upkdouble(&do_again, 1, 1);

  printf("\nProgram m1.c cwt = %d, ewt = %d is done\n", comm_gain, my_wt/4);

/* Program Finished exit PVM before stopping */
  pvm_exit();

} /** END OF MAIN PROGRAM **/
```

THE INDIVIDUAL SLAVE PROGRAMS:

```
/*****************************************************************************
Slave program for node P1B
*****************************************************************************/

#include "pvm3.h"
#include <stdio.h>
#include <sys/time.h>
#include <time.h>
#include <sys/types.h>
#include <signal.h>
#include <stdlib.h>

    /* CONSTANTS */
#define parent 6
#define child 8

main()
{
  int i,k,mytid, master;
  int tids[20];
  int nproc, msgtype, me;
  int rnum, rnumtot=0, done=0, disize, dosize;
  double data_mat[10000];
  int my_wt, comm_gain;

/* enroll in pvm */

  mytid = pvm_mytid();
  master = pvm_parent();

  pvm_recv(master, 20 );
  pvm_upkint(&nproc, 1, 1);
  pvm_upkint(tids, nproc, 1);
  pvm_upkint(&my_wt, 1, 1);
  pvm_upkint(&comm_gain, 1, 1);

  pvm_recv(master, 25 );
  pvm_upkint(&disize, 1, 1);
  pvm_upkint(&dosize, 1, 1);

  for ( i=0; i<nproc; i++ )
    if (mytid == tids[i] ) { me = i; break;}

  while (done == 0)
  {
    pvm_recv(tids[parent], me+1);
    pvm_upkdouble(data_mat, disize*comm_gain, 1);

    if (data_mat[0] < 0)
```

```
      done = 1;

/**   slave execution core **/

    for (k = 0; k < my_wt; k++)
      for (i = 0; i < 1360; i++)
      {
        rnum = rand();
        rnumtot = rnumtot + rnum;
        data_mat[i+1] = i + 1;
      }

    pvm_initsend(PvmDataDefault );
    pvm_pkdouble(data_mat, dosize*comm_gain, 1);
    pvm_send(tids[child], child+1);

  }    /* end of while done == 0 loop */

/* Program finished. Exit PVM before stopping */

  pvm_exit();

} /* End of Slave program p1b.c */
```

```
/**************************************************************************
Slave program for node P1C
**************************************************************************/

#include "pvm3.h"
#include <stdio.h>
#include <sys/time.h>
#include <time.h>
#include <sys/types.h>
#include <signal.h>
#include <stdlib.h>

    /* CONSTANTS */
#define parent 9
#define child 15

main()
{
  int i,k,mytid, master;
  int tids[20];
  int nproc, msgtype, me, disize, dosize;
  int rnum, rnumtot=0, done=0;
  double data_mat[10000];
  double do_again = 2.2;
  int my_wt, comm_gain;

/* enroll in pvm */

  mytid = pvm_mytid();
  master = pvm_parent();

  pvm_recv(master, 20 );
  pvm_upkint(&nproc, 1, 1);
  pvm_upkint(tids, nproc, 1);
  pvm_upkint(&my_wt, 1, 1);
  pvm_upkint(&comm_gain, 1, 1);

  pvm_recv(master, 25);
  pvm_upkint(&disize, 1, 1);
  pvm_upkint(&dosize, 1, 1);

  for ( i=0; i<nproc; i++ )
    if (mytid == tids[i] ) { me = i; break;}

  while (done == 0)
  {
    pvm_recv(tids[parent], me+1);
    pvm_upkdouble(data_mat, disize*comm_gain, 1);

    if (data_mat[0] < 0)
      done = 1;
```

```
/**  slave execution core **/

    for (k = 0; k < my_wt; k++)
      for (i = 0; i < 1360; i++)
      {
        rnum = rand();
        rnumtot = rnumtot + rnum;
        data_mat[i+1] = i + 1;
      }

    pvm_initsend(PvmDataDefault );
    pvm_pkdouble(data_mat, dosize*comm_gain, 1);
    pvm_send(tids[child], child+1);

    pvm_initsend(PvmDataDefault);
    pvm_pkdouble(&do_again, 1, 1);
    pvm_send(master, 25);

  }    /* end of while done == 0 loop */

/* Program finished. Exit PVM before stopping */

  pvm_exit();

} /* End of Slave program p1c.c */
```

```
/*************************************************************************
Slave program for node P2A
*************************************************************************/

#include "pvm3.h"
#include <stdio.h>
#include <sys/time.h>
#include <time.h>
#include <sys/types.h>
#include <signal.h>
#include <stdlib.h>

    /* CONSTANTS */
#define parent 12
#define child1 11
#define child2 6

main()
{
  int i,k,mytid, master, disize, dosize;
  int tids[20], my_wt, comm_gain;
  int nproc, msgtype, me;
  int rnum, rnumtot=0, done=0;
  double data_mat[10000];

/* enroll in pvm */

  mytid = pvm_mytid();
  master = pvm_parent();

  pvm_recv(master, 20 );
  pvm_upkint(&nproc, 1, 1);
  pvm_upkint(tids, nproc, 1);
  pvm_upkint(&my_wt, 1, 1);
  pvm_upkint(&comm_gain, 1, 1);

  pvm_recv(master, 25);
  pvm_upkint(&disize, 1, 1);
  pvm_upkint(&dosize, 1, 1);

  for ( i=0; i<nproc; i++ )
    if (mytid == tids[i] ) { me = i; break;}

  while (done == 0)
  {
    pvm_recv(tids[parent], me+1);
    pvm_upkdouble(data_mat, disize*comm_gain, 1);

    if (data_mat[0] < 0)
      done = 1;
```

```
/**  slave execution core **/

    for (k = 0; k < my_wt; k++)
      for (i = 0; i < 1360; i++)
      {
        rnum = rand();
        rnumtot = rnumtot + rnum;
        data_mat[i+1] = i + 1;
      }

    pvm_initsend(PvmDataDefault );
    pvm_pkdouble(data_mat, dosize*comm_gain, 1);
    pvm_send(tids[child1], child1+1);

    pvm_initsend(PvmDataDefault );
    pvm_pkdouble(data_mat, dosize*comm_gain, 1);
    pvm_send(tids[child2], child2+1);

  }    /* end of while done == 0 loop */

/* Program finished. Exit PVM before stopping */

  pvm_exit();

} /* End of Slave program p2a.c */
```

```
/*************************************************************************Slave program
for node P2D *************************************************************************/

#include "pvm3.h"
#include <stdio.h>
#include <sys/time.h>
#include <time.h>
#include <sys/types.h>
#include <signal.h>
#include <stdlib.h>

    /* CONSTANTS */
#define parent 3
#define child1 7
#define child2 10

main()
{
  int i,k,mytid, master;
  int tids[20], my_wt, comm_gain;
  int nproc, msgtype, me, disize, dosize;
  int rnum, rnumtot=0, done=0;
  double data_mat[10000];

/* enroll in pvm */

  mytid = pvm_mytid();
  master = pvm_parent();

  pvm_recv(master, 20 );
  pvm_upkint(&nproc, 1, 1);
  pvm_upkint(tids, nproc, 1);
  pvm_upkint(&my_wt, 1, 1);
  pvm_upkint(&comm_gain, 1, 1);

  pvm_recv(master, 25);
  pvm_upkint(&disize, 1, 1);
  pvm_upkint(&dosize, 1, 1);

  for ( i=0; i<nproc; i++ )
    if (mytid == tids[i] ) { me = i; break;}


  while (done == 0)
  {
    pvm_recv(tids[parent], me+1);
    pvm_upkdouble(data_mat, disize*comm_gain, 1);

    if (data_mat[0] < 0)
      done = 1;
```

```
/**  slave execution core **/

    for (k = 0; k < my_wt; k++)
      for (i = 0; i < 1360; i++)
      {
        rnum = rand();
        rnumtot = rnumtot + rnum;
        data_mat[i+1] = i + 1;
      }

    pvm_initsend(PvmDataDefault );
    pvm_pkdouble(data_mat, dosize*comm_gain, 1);
    pvm_send(tids[child1], child1+1);

    pvm_initsend(PvmDataDefault );
    pvm_pkdouble(data_mat, dosize*comm_gain, 1);
    pvm_send(tids[child2], child2+1);

  }    /* end of while done == 0 loop */

/* Program finished. Exit PVM before stopping */

  pvm_exit();


} /* End of Slave program p2d.c */
```

```
/*************************************************************************
Slave program for node P3A
*************************************************************************/

#include "pvm3.h"
#include <stdio.h>
#include <sys/time.h>
#include <time.h>
#include <sys/types.h>
#include <signal.h>
#include <stdlib.h>

    /* CONSTANTS */
#define parent 2
#define child 0

main()
{
  int i,k,mytid, master;
  int tids[20], my_wt, comm_gain, disize, dosize;
  int nproc, msgtype, me;
  int rnum, rnumtot=0, done=0;
  double data_mat[10000];

/* enroll in pvm */

  mytid = pvm_mytid();
  master = pvm_parent();

  pvm_recv(master, 20 );
  pvm_upkint(&nproc, 1, 1);
  pvm_upkint(tids, nproc, 1);
  pvm_upkint(&my_wt, 1, 1);
  pvm_upkint(&comm_gain, 1, 1);

  pvm_recv(master, 25 );
  pvm_upkint(&disize, 1, 1);
  pvm_upkint(&dosize, 1, 1);

  for ( i=0; i<nproc; i++ )
    if (mytid == tids[i] ) { me = i; break;}


  while (done == 0)
  {
    pvm_recv(tids[parent], me+1);
    pvm_upkdouble(data_mat, disize*comm_gain, 1);

    if (data_mat[0] < 0)
      done = 1;
```

```
/**  slave execution core **/

    for (k = 0; k < my_wt; k++)
      for (i = 0; i < 1360; i++)
      {
        rnum = rand();
        rnumtot = rnumtot + rnum;
        data_mat[i+1] = i + 1;
      }

    pvm_initsend(PvmDataDefault );
    pvm_pkdouble(data_mat, dosize*comm_gain, 1);
    pvm_send(tids[child], child+1);

  }    /* end of while done == 0 loop */

/* Program finished. Exit PVM before stopping */

  pvm_exit();


} /* End of Slave program p3a.c */
```

```
/*****************************************************************************
*  Slave program for node P3B                                                *
*****************************************************************************/

#include "pvm3.h"
#include <stdio.h>
#include <sys/time.h>
#include <time.h>
#include <sys/types.h>
#include <signal.h>
#include <stdlib.h>

    /* CONSTANTS */
#define parent 5
#define child 14

main()
{
  int i,k,mytid, master, disize, dosize;
  int tids[20], my_wt, comm_gain;
  int nproc, msgtype, me;
  int rnum, rnumtot=0, done=0;
  double data_mat[10000];

/* enroll in pvm */

  mytid = pvm_mytid();
  master = pvm_parent();

  pvm_recv(master, 20 );
  pvm_upkint(&nproc, 1, 1);
  pvm_upkint(tids, nproc, 1);
  pvm_upkint(&my_wt, 1, 1);
  pvm_upkint(&comm_gain, 1, 1);

  pvm_recv(master, 25 );
  pvm_upkint(&disize, 1, 1);
  pvm_upkint(&dosize, 1, 1);


  for ( i=0; i<nproc; i++ )
    if (mytid == tids[i] ) { me = i; break;}


  while (done == 0)
  {
    pvm_recv(tids[parent], me+1);
    pvm_upkdouble(data_mat, disize*comm_gain, 1);

    if (data_mat[0] < 0)
      done = 1;
```

```
/**  slave execution core **/

    for (k = 0; k < my_wt; k++)
      for (i = 0; i < 1360; i++)
      {
        rnum = rand();
        rnumtot = rnumtot + rnum;
        data_mat[i+1] = i + 1;
      }

    pvm_initsend(PvmDataDefault );
    pvm_pkdouble(data_mat, dosize*comm_gain, 1);
    pvm_send(tids[child], child+1);

  }    /* end of while done == 0 loop */

/* Program finished. Exit PVM before stopping */

  pvm_exit();


} /* End of Slave program p3b.c */
```

```
/*************************************************************************
Slave program for node P3C
*************************************************************************/

#include "pvm3.h"
#include <stdio.h>
#include <sys/time.h>
#include <time.h>
#include <sys/types.h>
#include <signal.h>
#include <stdlib.h>

    /* CONSTANTS */
#define parent1 11
#define parent2 0
#define child 15

main()
{
  int i,k,mytid, master;
  int tids[20], my_wt, comm_gain;
  int nproc, msgtype, me;
  int rnum, rnumtot=0, done=0;
  double data_mat[10000];
  int disize, disize2, dosize;

/* enroll in pvm */

  mytid = pvm_mytid();
  master = pvm_parent();

  pvm_recv(master, 20 );
  pvm_upkint(&nproc, 1, 1);
  pvm_upkint(tids, nproc, 1);
  pvm_upkint(&my_wt, 1, 1);
  pvm_upkint(&comm_gain, 1, 1);

  pvm_recv(master, 25);
  pvm_upkint(&disize, 1, 1);
  pvm_upkint(&disize2, 1, 1);
  pvm_upkint(&dosize, 1, 1);

  my_wt = my_wt*2;

  for ( i=0; i<nproc; i++ )
    if (mytid == tids[i] ) { me = i; break;}

  while (done == 0)
  {
    pvm_recv(tids[parent1], me+1);
    pvm_upkdouble(data_mat, disize*comm_gain, 1);
```

```
    pvm_recv(tids[parent2], me+1);
    pvm_upkdouble(data_mat, disize2*comm_gain, 1);

    if (data_mat[0] < 0)
      done = 1;

/**  slave execution core **/

    for (k = 0; k < my_wt; k++)
      for (i = 0; i < 1360; i++)
      {
        rnum = rand();
        rnumtot = rnumtot + rnum;
        data_mat[i+1] = i + 1;
      }

    pvm_initsend(PvmDataDefault );
    pvm_pkdouble(data_mat, dosize*comm_gain, 1);
    pvm_send(tids[child], child+1);

  }    /* end of while done == 0 loop */

/* Program finished. Exit PVM before stopping */

  pvm_exit();

} /* End of Slave program p3c.c */
```

```
/**************************************************************************
Slave program for node P4A
**************************************************************************/

#include "pvm3.h"
#include <stdio.h>
#include <sys/time.h>
#include <time.h>
#include <sys/types.h>
#include <signal.h>
#include <stdlib.h>

    /* CONSTANTS */
#define parent 13
#define child 1

main()
{
  int i,k,mytid, master;
  int tids[20], my_wt, comm_gain;
  int nproc, msgtype, me, disize, dosize;
  int rnum, rnumtot=0, done=0;
  double data_mat[10000];

/* enroll in pvm */

  mytid = pvm_mytid();
  master = pvm_parent();

  pvm_recv(master, 20 );
  pvm_upkint(&nproc, 1, 1);
  pvm_upkint(tids, nproc, 1);
  pvm_upkint(&my_wt, 1, 1);
  pvm_upkint(&comm_gain, 1, 1);

  pvm_recv(master, 25);
  pvm_upkint(&disize, 1, 1);
  pvm_upkint(&dosize, 1, 1);

  for ( i=0; i<nproc; i++ )
    if (mytid == tids[i] ) { me = i; break;}


  while (done == 0)
  {
    pvm_recv(tids[parent], me+1);
    pvm_upkdouble(data_mat, disize*comm_gain, 1);

    if (data_mat[0] < 0)
      done = 1;
```

```
/**  slave execution core **/

    for (k = 0; k < my_wt; k++)
      for (i = 0; i < 1360; i++)
      {
        rnum = rand();
        rnumtot = rnumtot + rnum;
        data_mat[i+1] = i + 1;
      }

    pvm_initsend(PvmDataDefault );
    pvm_pkdouble(data_mat, dosize*comm_gain, 1);
    pvm_send(tids[child], child+1);

  }    /* end of while done == 0 loop */

/* Program finished. Exit PVM before stopping */

  pvm_exit();

} /* End of Slave program p4a.c */
```

```
/*************************************************************************
Slave program for node P4B
*************************************************************************/

#include "pvm3.h"
#include <stdio.h>
#include <sys/time.h>
#include <time.h>
#include <sys/types.h>
#include <signal.h>
#include <stdlib.h>

    /* CONSTANTS */
#define parent 5
#define child 4


main()
{
  int i,k,mytid, master;
  int tids[20], my_wt, comm_gain;
  int nproc, msgtype, me, disize, dosize;
  int rnum, rnumtot=0, done=0;
  double data_mat[10000];

/* enroll in pvm */

  mytid = pvm_mytid();
  master = pvm_parent();

  pvm_recv(master, 20 );
  pvm_upkint(&nproc, 1, 1);
  pvm_upkint(tids, nproc, 1);
  pvm_upkint(&my_wt, 1, 1);
  pvm_upkint(&comm_gain, 1, 1);

  pvm_recv(master, 25 );
  pvm_upkint(&disize, 1, 1);
  pvm_upkint(&dosize, 1, 1);


  for ( i=0; i<nproc; i++ )
    if (mytid == tids[i] ) { me = i; break;}


  while (done == 0)
  {
    pvm_recv(tids[parent], me+1);
    pvm_upkdouble(data_mat, disize*comm_gain, 1);

    if (data_mat[0] < 0)
```

```
      done = 1;

/**   slave execution core **/

    for (k = 0; k < my_wt; k++)
      for (i = 0; i < 1360; i++)
      {
        rnum = rand();
        rnumtot = rnumtot + rnum;
        data_mat[i+1] = i + 1;
      }

    pvm_initsend(PvmDataDefault );
    pvm_pkdouble(data_mat, dosize*comm_gain, 1);
    pvm_send(tids[child], child+1);

  }    /* end of while done == 0 loop */

/* Program finished. Exit PVM before stopping */

  pvm_exit();


} /* End of Slave program p4b.c */
```

```
/*************************************************************************
Slave program for node P4C
*************************************************************************/

#include "pvm3.h"
#include <stdio.h>
#include <sys/time.h>
#include <time.h>
#include <sys/types.h>
#include <signal.h>
#include <stdlib.h>

    /* CONSTANTS */
#define parent 2
#define child 8

main()
{
  int i,k,mytid, master;
  int tids[20], my_wt, comm_gain;
  int nproc, msgtype, me, disize, dosize;
  int rnum, rnumtot=0, done=0;
  double data_mat[10000];

/* enroll in pvm */

  mytid = pvm_mytid();
  master = pvm_parent();

  pvm_recv(master, 20 );
  pvm_upkint(&nproc, 1, 1);
  pvm_upkint(tids, nproc, 1);
  pvm_upkint(&my_wt, 1, 1);
  pvm_upkint(&comm_gain, 1, 1);

  pvm_recv(master, 25 );
  pvm_upkint(&disize, 1, 1);
  pvm_upkint(&dosize, 1, 1);

  for ( i=0; i<nproc; i++ )
    if (mytid == tids[i] ) { me = i; break;}


  while (done == 0)
  {
    pvm_recv(tids[parent], me+1);
    pvm_upkdouble(data_mat, disize*comm_gain, 1);

    if (data_mat[0] < 0)
      done = 1;
```

```
/**  slave execution core **/

    for (k = 0; k < my_wt; k++)
      for (i = 0; i < 1360; i++)
      {
        rnum = rand();
        rnumtot = rnumtot + rnum;
        data_mat[i+1] = i + 1;
      }

    pvm_initsend(PvmDataDefault );
    pvm_pkdouble(data_mat, dosize*comm_gain, 1);
    pvm_send(tids[child], child+1);

  }    /* end of while done == 0 loop */

/* Program finished. Exit PVM before stopping */

  pvm_exit();


} /* End of Slave program p4c.c */
```

```
/*************************************************************************
Slave program for node P4D
*************************************************************************/

#include "pvm3.h"
#include <stdio.h>
#include <sys/time.h>
#include <time.h>
#include <sys/types.h>
#include <signal.h>
#include <stdlib.h>

    /* CONSTANTS */
#define child 2

main()
{
  int i,k,mytid, master;
  int tids[20], my_wt, comm_gain;
  int nproc, msgtype, me, disize, dosize;
  int rnum, rnumtot=0, done=0;
  double data_mat[10000];

/* enroll in pvm */

  mytid = pvm_mytid();
  master = pvm_parent();

  pvm_recv(master, 20 );
  pvm_upkint(&nproc, 1, 1);
  pvm_upkint(tids, nproc, 1);
  pvm_upkint(&my_wt, 1, 1);
  pvm_upkint(&comm_gain, 1, 1);

  pvm_recv(master, 25);
  pvm_upkint(&disize, 1, 1);
  pvm_upkint(&dosize, 1, 1);

  for ( i=0; i<nproc; i++ )
    if (mytid == tids[i] ) { me = i; break;}

  while (done == 0)
  {
    pvm_recv(master, me+1);
    pvm_upkdouble(data_mat, disize*comm_gain, 1);

    if (data_mat[0] < 0)
      done = 1;

/**  slave execution core **/
```

```
    for (k = 0; k < my_wt; k++)
      for (i = 0; i < 1360; i++)
      {
        rnum = rand();
        rnumtot = rnumtot + rnum;
        data_mat[i+1] = i + 1;
      }

    pvm_initsend(PvmDataDefault );
    pvm_pkdouble(data_mat, dosize*comm_gain, 1);
    pvm_send(tids[child], child+1);

  }    /* end of while done == 0 loop */

/* Program finished. Exit PVM before stopping */

  pvm_exit();


} /* End of Slave program p4d.c */
```

```
/*************************************************************************
Slave program for node P5A
*************************************************************************/

#include "pvm3.h"
#include <stdio.h>
#include <sys/time.h>
#include <time.h>
#include <sys/types.h>
#include <signal.h>
#include <stdlib.h>

    /* CONSTANTS */
#define child 9

main()
{
  int i,k,mytid, master;
  int tids[20], my_wt, comm_gain;
  int nproc, msgtype, me, disize, dosize;
  int rnum, rnumtot=0, done=0;
  double data_mat[10000];

/* enroll in pvm */

  mytid = pvm_mytid();
  master = pvm_parent();

  pvm_recv(master, 20 );
  pvm_upkint(&nproc, 1, 1);
  pvm_upkint(tids, nproc, 1);
  pvm_upkint(&my_wt, 1, 1);
  pvm_upkint(&comm_gain, 1, 1);

  pvm_recv(master, 25);
  pvm_upkint(&disize, 1, 1);
  pvm_upkint(&dosize, 1, 1);

  for ( i=0; i<nproc; i++ )
    if (mytid == tids[i] ) { me = i; break;}


  while (done == 0)
  {
    pvm_recv(master, me+1);
    pvm_upkdouble(data_mat, disize*comm_gain, 1);

    if (data_mat[0] < 0)
      done = 1;

 /**  slave execution core **/
```

```
    for (k = 0; k < my_wt; k++)
      for (i = 0; i < 1360; i++)
      {
        rnum = rand();
        rnumtot = rnumtot + rnum;
        data_mat[i+1] = i + 1;
      }

    pvm_initsend(PvmDataDefault );
    pvm_pkdouble(data_mat, dosize*comm_gain, 1);
    pvm_send(tids[child], child+1);

  }    /* end of while done == 0 loop */

/* Program finished. Exit PVM before stopping */

  pvm_exit();


} /* End of Slave program p5a.c */
```

```
/**********************************************************************
Slave program for node P5B
**********************************************************************/

#include "pvm3.h"
#include <stdio.h>
#include <sys/time.h>
#include <time.h>
#include <sys/types.h>
#include <signal.h>
#include <stdlib.h>

    /* CONSTANTS */
#define parent 7
#define child 4


main()
{
  int i,k,mytid, master;
  int tids[20], my_wt, comm_gain;
  int nproc, msgtype, me, dosize, disize;
  int rnum, rnumtot=0, done=0;
  double data_mat[10000];

/* enroll in pvm */

  mytid = pvm_mytid();
  master = pvm_parent();

  pvm_recv(master, 20 );
  pvm_upkint(&nproc, 1, 1);
  pvm_upkint(tids, nproc, 1);
  pvm_upkint(&my_wt, 1, 1);
  pvm_upkint(&comm_gain, 1, 1);

  pvm_recv(master, 25 );
  pvm_upkint(&disize, 1, 1);
  pvm_upkint(&dosize, 1, 1);

  for ( i=0; i<nproc; i++ )
    if (mytid == tids[i] ) { me = i; break;}


  while (done == 0)
  {
    pvm_recv(tids[parent], me+1);
    pvm_upkdouble(data_mat, disize*comm_gain, 1);

    if (data_mat[0] < 0)
      done = 1;
```

```
/**  slave execution core **/

    for (k = 0; k < my_wt; k++)
      for (i = 0; i < 1360; i++)
      {
        rnum = rand();
        rnumtot = rnumtot + rnum;
        data_mat[i+1] = i + 1;
      }

    pvm_initsend(PvmDataDefault );
    pvm_pkdouble(data_mat, dosize*comm_gain, 1);
    pvm_send(tids[child], child+1);

  }    /* end of while done == 0 loop */

/* Program finished. Exit PVM before stopping */

  pvm_exit();


} /* End of Slave program p5b.c */
```

```
/*************************************************************************
Slave program for node P5C
*************************************************************************/
#include "pvm3.h"
#include <stdio.h>
#include <sys/time.h>
#include <time.h>
#include <sys/types.h>
#include <signal.h>
#include <stdlib.h>

    /* CONSTANTS */
#define parent1 1
#define parent2 4
#define parent3 8

main()
{
  int i,k,mytid, master, disize, disize2, disize3;
  int tids[20], my_wt, comm_gain;
  int nproc, msgtype, me, go_now = 1;
  int rnum, rnumtot=0, done=0;
  double data_mat[10000];
  int oldintv = 0, newintv = 0, tcalc;
  FILE *ofp;
  struct timeval stime;

/* enroll in pvm */
  mytid = pvm_mytid();
  master = pvm_parent();

  pvm_recv(master, 20 );
  pvm_upkint(&nproc, 1, 1);
  pvm_upkint(tids, nproc, 1);
  pvm_upkint(&my_wt, 1, 1);
  pvm_upkint(&comm_gain, 1, 1);
  my_wt = my_wt*2;
  pvm_recv(master, 25 );
  pvm_upkint(&disize, 1, 1);
  pvm_upkint(&disize2, 1, 1);
  pvm_upkint(&disize3, 1, 1);

  for ( i=0; i<nproc; i++ )
    if (mytid == tids[i] ) { me = i; break;}

  ofp = fopen("/home3/stone/Thesis/matlab_files/No_sched.out","w");

  while (done == 0)
  {
    pvm_recv(tids[parent1], me+1);
    pvm_upkdouble(data_mat, disize*comm_gain, 1);
```

```
    pvm_recv(tids[parent2], me+1);
    pvm_upkdouble(data_mat, disize2*comm_gain, 1);

    pvm_recv(tids[parent3], me+1);
    pvm_upkdouble(data_mat, disize3*comm_gain, 1);

    if (data_mat[0] < 0)
      done = 1;

/**  slave execution core **/
    for (k = 0; k < my_wt; k++)
      for (i = 0; i < 1360; i++)
      {
        rnum = rand();
        rnumtot = rnumtot + rnum;
        data_mat[i+1] = i + 1;
      }

    gettimeofday(&stime, (struct timeval*)0);
    newintv=stime.tv_sec*1000000+stime.tv_usec;
    tcalc = newintv-oldintv;
    fprintf(ofp,"\n%d",tcalc);
    oldintv = newintv;
  }    /* end of while done == 0 loop */
  fclose(ofp);
/* Tell the Master all slaves have terminated */
    pvm_initsend(PvmDataDefault );
    pvm_pkdouble(&go_now, 1, 1);
    pvm_send(master, 35);
/* Program finished. Exit PVM before stopping */
  pvm_exit();
} /* End of Slave program p5c.c */
```

APPENDIX E - SCHEDULED NODE PROCESSING PROGRAMS

MASTER PROGRAM (PROCESSOR 1):

```
/*****************************************************************************
Scheduled master program, also handles node Processor 1 communication and execution
requirements.
*****************************************************************************/
#include "pvm3.h"
#include <stdio.h>
#include <sys/time.h>
#include <time.h>
#include <sys/types.h>
#include <signal.h>
#include <stdlib.h>

    /* CONSTANTS */
#define done_lp  1000          /* Loop iteration counter */
#define snl 400                /* Iteration number for start of network loading */

    /* GRAPH TIME VARIABLES */
int  ld_num = 55000;
int  p1top2 = 300;      /* These variables contain the interprocessor comm*/
int  p1top3 = 300;      /* costs, read has Processor i to Processor j */
int  p1top4 = 300;
int  p1top5 = 300 + 300;
int  p2top3 = 350 + 350;
int  p2top4 = 350 + 350;
int  p2top5 = 300;
int  p3top1 = 525;
int  p3top5 = 300 + 525;
int  p4top1 = 350;
int  p4top2 = 400 + 350;
int  p4top3 = 350;
int  p5top2 = 300;
int  p5top4 = 900;
int comm_gain;          /* For varying communication weights */
int my_wt ;                     /* For varying execution weights */


    /* GLOBAL VARIABLES */
int nproc = 4;
int mytid;                  /* my task id */
int tids[20];               /* slave task ids */
int done = 0, who;
double data_mat[9000], go_now;

main()
{
  char SLAVENAME[3];
```

```
  int i, k;
  int rnum, rnumtot;
  char myname[5];
  struct itimerval tmrval;
  int nnproc = 3, stids[5], wnl = 0, nl_done = 0;

  printf("\nComm wt =  ");
  scanf("%d", &comm_gain);
  printf("\Ex wt =  ");
  scanf("%d", &my_wt);
  my_wt = my_wt*4;

  printf("\With NET loading type 1, without NET loading type 2:  ");
  scanf("%d", &wnl);

/* initialize matices */

  for(k=0; k<1500; k++)
    data_mat[k]=(double)k+5.66666;

/* enroll in pvm */

  mytid = pvm_mytid();

/* start up slave tasks */

  gethostname(myname,5);

  pvm_spawn("p2", NULL, 1, "sun3", 1, &tids[0]);
  pvm_spawn("p3", NULL, 1, "sun8", 1, &tids[1]);
  pvm_spawn("p4", NULL, 1, "sun9", 1, &tids[2]);
  pvm_spawn("p5", NULL, 1, "sun20", 1, &tids[3]);

  pvm_initsend(PvmDataDefault);
  pvm_pkint(&nproc, 1, 1);
  pvm_pkint(tids, nproc, 1);
  pvm_pkint(&my_wt, 1, 1);
  pvm_pkint(&comm_gain, 1, 1);
  pvm_mcast(tids, nproc, 10);

  pvm_initsend(PvmDataDefault );
  pvm_pkint(&p1top2, 1, 1);
  pvm_pkint(&p4top2, 1, 1);
  pvm_pkint(&p5top2, 1, 1);
  pvm_pkint(&p2top3, 1, 1);
  pvm_pkint(&p2top4, 1, 1);
  pvm_pkint(&p2top5, 1, 1);
  pvm_send(tids[0], 20);

  pvm_initsend(PvmDataDefault );
  pvm_pkint(&p1top3, 1, 1);
```

```
pvm_pkint(&p2top3, 1, 1);
pvm_pkint(&p4top3, 1, 1);
pvm_pkint(&p3top1, 1, 1);
pvm_pkint(&p3top5, 1, 1);
pvm_send(tids[1], 20);

pvm_initsend(PvmDataDefault );
pvm_pkint(&p1top4, 1, 1);
pvm_pkint(&p2top4, 1, 1);
pvm_pkint(&p5top4, 1, 1);
pvm_pkint(&p4top1, 1, 1);
pvm_pkint(&p4top2, 1, 1);
pvm_pkint(&p4top3, 1, 1);
pvm_send(tids[2], 20);

pvm_initsend(PvmDataDefault );
pvm_pkint(&p1top5, 1, 1);
pvm_pkint(&p2top5, 1, 1);
pvm_pkint(&p3top5, 1, 1);
pvm_pkint(&p5top2, 1, 1);
pvm_pkint(&p5top4, 1, 1);
pvm_send(tids[3], 20);

if (wnl == 1)
{
  pvm_spawn("s1", NULL, 1, "sun3", 1, &stids[0]);
  pvm_spawn("s2", NULL, 1, "sun20", 1, &stids[1]);
  pvm_spawn("s3", NULL, 1, "sun8", 1, &stids[2]);

  pvm_initsend(PvmDataDefault);
  pvm_pkint(&snproc, 1, 1);
  pvm_pkint(stids, snproc, 1);
  pvm_mcast(stids, snproc, 62);
}

/* Begin User Program */

for (done = 0; done < done_lp; done ++)
{

  if (done == snl && wnl == 1)
  {
    pvm_initsend(PvmDataDefault);
    pvm_pkint(&ld_num, 1, 1);
    pvm_send(stids[0], 22);
  }

  if (done == done_lp - 1)
    data_mat[0] = -444.555;

  pvm_initsend(PvmDataDefault );
```

```
    pvm_pkdouble(data_mat, p1top5*comm_gain, 1);
    pvm_send(tids[3], 4);

    pvm_initsend(PvmDataDefault );
    pvm_pkdouble(data_mat, p1top4*comm_gain, 1);
    pvm_send(tids[2], 3);

    pvm_initsend(PvmDataDefault );
    pvm_pkdouble(data_mat, p1top3*comm_gain, 1);
    pvm_send(tids[1], 2);

    pvm_initsend(PvmDataDefault );
    pvm_pkdouble(data_mat, p1top2*comm_gain, 1);
    pvm_send(tids[0], 1);

/* Execution core section */
    for (k = 0; k < my_wt*2; k++)
      for (i = 0; i < 1360; i++)
      {
        rnum = rand();
        rnumtot = rnumtot + rnum;
        data_mat[i+1] = i + 1;
      }

    for (k = 0; k < my_wt; k++)
      for (i = 0; i < 1360; i++)
      {
        rnum = rand();
        rnumtot = rnumtot + rnum;
        data_mat[i+1] = i + 1;
      }

    for (k = 0; k < my_wt; k++)
      for (i = 0; i < 1360; i++)
      {
        rnum = rand();
        rnumtot = rnumtot + rnum;
        data_mat[i+1] = i + 1;
      }

    printf("\nOn loop number %d\n",done);

    if(data_mat[0] >= 0)
    {
      pvm_recv(tids[1], 2);
      pvm_upkdouble(data_mat, p3top1*comm_gain, 1);

      pvm_recv(tids[2], 3);
      pvm_upkdouble(data_mat, p4top1*comm_gain, 1);

      pvm_recv(tids[3], 4);
```

```
      pvm_upkdouble(&go_now, 1, 1);
    }

  }    /* end of for loop */

  printf("\nThe loop is done\n");

/* Ensure all slaves have quit prior to termination */
  for(i=0; i<nproc; i++)
  {
    pvm_recv(-1, 35);
    pvm_upkint(&who, 1, 1);
  }
 printf("\nProgram m2.c cwt = %d, ewt = %d is done\n", comm_gain, my_wt/4);

/* Program Finished exit PVM before stopping */
  pvm_exit();
} /** END OF MAIN PROGRAM **/
```

THE INDIVIDUAL SLAVE PROGRAMS:

```
/**********************************************************************
Slave program for scheduled Processor 2
**********************************************************************/
#include "pvm3.h"
#include <stdio.h>
#include <sys/time.h>
#include <time.h>
#include <sys/types.h>
#include <signal.h>
#include <stdlib.h>

    /* CONSTANTS */
#define parent1 2
#define parent2 3

main()
{
  int i,k,mytid, master;
  int tids[20], stids[20], my_wt, comm_gain;
  int nproc, msgtype, me, snproc;
  int rnum, rnumtot=0, done=0;
  double data_mat[10000];
/* These hold the input cost,"i",  or the output cost,"o" */
  int disize1, disize4, disize5, dosize3, dosize4, dosize5;

/* enroll in pvm */

  mytid = pvm_mytid();
  master = pvm_parent();

  pvm_recv(master, 10 );
  pvm_upkint(&nproc, 1, 1);
  pvm_upkint(tids, nproc, 1);
  pvm_upkint(&my_wt, 1, 1);
  pvm_upkint(&comm_gain, 1, 1);

  pvm_recv(master, 20 );
  pvm_upkint(&disize1, 1, 1);
  pvm_upkint(&disize4, 1, 1);
  pvm_upkint(&disize5, 1, 1);
  pvm_upkint(&dosize3, 1, 1);
  pvm_upkint(&dosize4, 1, 1);
  pvm_upkint(&dosize5, 1, 1);

  for ( i=0; i<nproc; i++ )
    if (mytid == tids[i] ) { me = i; break;}

  pvm_recv(master, me+1);
  pvm_upkdouble(data_mat, disize1*comm_gain, 1);
```

```
while (done == 0)
{
  pvm_initsend(PvmDataDefault );
  pvm_pkdouble(data_mat, dosize5*comm_gain, 1);
  pvm_send(tids[3], 4);

  pvm_initsend(PvmDataDefault );
  pvm_pkdouble(data_mat, dosize4*comm_gain, 1);
  pvm_send(tids[2], 3);

  pvm_initsend(PvmDataDefault );
  pvm_pkdouble(data_mat, dosize3*comm_gain, 1);
  pvm_send(tids[1], 2);

/**  slave execution core **/

  for (k = 0; k < my_wt; k++)          /* simulates node A */
    for (i = 0; i < 1360; i++)
    {
      rnum = rand();
      rnumtot = rnumtot + rnum;
      data_mat[i+1] = i + 1;
    }

  for (k = 0; k < my_wt; k++)          /* simulates node B */
    for (i = 0; i < 1360; i++)
    {
      rnum = rand();
      rnumtot = rnumtot + rnum;
      data_mat[i+1] = i + 1;
    }
                                       /* simulates node C */
  for (k = 0; k < my_wt; k++)
    for (i = 0; i < 1360; i++)
    {
      rnum = rand();
      rnumtot = rnumtot + rnum;
      data_mat[i+1] = i + 1;
    }

  for (k = 0; k < my_wt; k++)          /* simulates node D */
    for (i = 0; i < 1360; i++)
    {
      rnum = rand();
      rnumtot = rnumtot + rnum;
      data_mat[i+1] = i + 1;
    }

  pvm_recv(tids[2], me+1);
  pvm_upkdouble(data_mat, disize4*comm_gain, 1);
```

```
    pvm_recv(tids[3], me+1);
    pvm_upkdouble(data_mat, disize5*comm_gain, 1);

    pvm_recv(master, me+1);
    pvm_upkdouble(data_mat, disize1*comm_gain, 1);

    if (data_mat[0] < 0)
    {
      done = 1;
      pvm_initsend(PvmDataDefault );
      pvm_pkdouble(data_mat, dosize3*comm_gain, 1);
      pvm_send(tids[1], 2);
    }
  }   /* end of while done == 0 loop */

/* Inform the master I have terminated */
  pvm_initsend(PvmDataDefault );
  pvm_pkint(&me, 1, 1);
  pvm_send(master, 35);

/* Program finished. Exit PVM before stopping */
  pvm_exit();

} /* End of Slave program p2.c */
```

```
/*************************************************************************
Slave program for scheduled processor 3
*************************************************************************/
#include "pvm3.h"
#include <stdio.h>
#include <sys/time.h>
#include <time.h>
#include <sys/types.h>
#include <signal.h>
#include <stdlib.h>

    /* CONSTANTS */
#define parent1 3
#define done_tag 45

main()
{
  int i,k,mytid, master;
  int tids[20], stids[20], my_wt, comm_gain;
  int nproc, msgtype, me, snproc;
  int rnum, rnumtot=0, done=0;
  double data_mat[10000];
  double go_now = 55.55;
  int disize1, disize2, disize4, dosize1, dosize5;

/* enroll in pvm */

  mytid = pvm_mytid();
  master = pvm_parent();

  pvm_recv(master, 10 );
  pvm_upkint(&nproc, 1, 1);
  pvm_upkint(tids, nproc, 1);
  pvm_upkint(&my_wt, 1, 1);
  pvm_upkint(&comm_gain, 1, 1);

  pvm_recv(master, 20 );
  pvm_upkint(&disize1, 1, 1);
  pvm_upkint(&disize2, 1, 1);
  pvm_upkint(&disize4, 1, 1);
  pvm_upkint(&dosize1, 1, 1);
  pvm_upkint(&dosize5, 1, 1);

  for ( i=0; i<nproc; i++ )
    if (mytid == tids[i] ) { me = i; break;}

  pvm_recv(tids[0], 2);
  pvm_upkdouble(data_mat, disize2*comm_gain, 1);

  while (done == 0)
  {
```

```
    pvm_initsend(PvmDataDefault );
    pvm_pkdouble(data_mat, dosize5*comm_gain, 1);
    pvm_send(tids[3], 4);

    pvm_initsend(PvmDataDefault );
    pvm_pkdouble(data_mat, dosize1*comm_gain, 1);
    pvm_send(master, me+1);

    pvm_initsend(PvmDataDefault );
    pvm_pkdouble(&go_now, 1, 1);
    pvm_send(tids[2], 20);

/**  slave execution core **/
    for (k = 0; k < my_wt; k++)          /* simulates node A */
      for (i = 0; i < 1360; i++)
      {
        rnum = rand();
        rnumtot = rnumtot + rnum;
        data_mat[i+1] = i + 1;
      }

    for (k = 0; k < my_wt; k++)          /* simulates node B */
      for (i = 0; i < 1360; i++)
      {
        rnum = rand();
        rnumtot = rnumtot + rnum;
        data_mat[i+1] = i + 1;
      }
                              /* simulates node C */
    for (k = 0; k < my_wt*2; k++)
      for (i = 0; i < 1360; i++)
      {
        rnum = rand();
        rnumtot = rnumtot + rnum;
        data_mat[i+1] = i + 1;
      }

    pvm_recv(master, me+1);
    pvm_upkdouble(data_mat, disize1*comm_gain, 1);

    pvm_recv(tids[2], me+1);
    pvm_upkdouble(data_mat, disize4*comm_gain, 1);

    pvm_recv(tids[0], me+1);
    pvm_upkdouble(data_mat, disize2*comm_gain, 1);

    if (data_mat[0] < 0)
    {
      done = 1;
      go_now = -34.33;
      pvm_initsend(PvmDataDefault);
```

```
        pvm_pkdouble(&go_now, 1, 1);
        pvm_send(tids[2], 20);
      }
    }   /* end of while done == 0 loop */

/* Inform the master I have terminated */
  pvm_initsend(PvmDataDefault );
  pvm_pkint(&me, 1, 1);
  pvm_send(master, 35);

/* Program finished. Exit PVM before stopping */
  pvm_exit();

} /* End of Slave program p3.c */
```

```
/*****************************************************************************
Slave program for scheduled processor 4
*****************************************************************************/
#include "pvm3.h"
#include <stdio.h>
#include <sys/time.h>
#include <time.h>
#include <sys/types.h>
#include <signal.h>
#include <stdlib.h>

    /* CONSTANTS */
#define parent1 2
#define parent2 3

main()
{
  int i,k,mytid, master;
  int tids[20], my_wt, comm_gain;
  int nproc, msgtype, me, snproc;
  int rnum, rnumtot=0, done=0;
  double data_mat[10000];
  double go_now;
  int disize1, disize2, disize5, dosize1, dosize2, dosize3;

/* enroll in pvm */
  mytid = pvm_mytid();
  master = pvm_parent();

  pvm_recv(master, 10 );
  pvm_upkint(&nproc, 1, 1);
  pvm_upkint(tids, nproc, 1);
  pvm_upkint(&my_wt, 1, 1);
  pvm_upkint(&comm_gain, 1, 1);

  pvm_recv(master, 20 );
  pvm_upkint(&disize1, 1, 1);
  pvm_upkint(&disize2, 1, 1);
  pvm_upkint(&disize5, 1, 1);
  pvm_upkint(&dosize1, 1, 1);
  pvm_upkint(&dosize2, 1, 1);
  pvm_upkint(&dosize3, 1, 1);

  for ( i=0; i<nproc; i++ )
    if (mytid == tids[i] ) { me = i; break;}

  data_mat[0] = 456.3333;

  pvm_recv(tids[1], 20);
  pvm_upkdouble(&go_now, 1, 1);
```

```
while (done == 0)
{
  pvm_initsend(PvmDataDefault );
  pvm_pkdouble(data_mat, dosize1*comm_gain, 1);
  pvm_send(master, me+1);

  pvm_initsend(PvmDataDefault );
  pvm_pkdouble(data_mat, dosize2*comm_gain, 1);
  pvm_send(tids[0], 1);

  pvm_initsend(PvmDataDefault );
  pvm_pkdouble(data_mat, dosize3*comm_gain, 1);
  pvm_send(tids[1], 2);

  pvm_initsend(PvmDataDefault );
  pvm_pkdouble(&go_now, 1, 1);
  pvm_send(tids[3], 4);

  if (data_mat[0] < 0)
    done = 1;

/**  slave execution core **/
  for (k = 0; k < my_wt; k++)          /* simulates node A */
    for (i = 0; i < 1360; i++)
    {
      rnum = rand();
      rnumtot = rnumtot + rnum;
      data_mat[i+1] = i + 1;
    }

  for (k = 0; k < my_wt; k++)          /* simulates node B */
    for (i = 0; i < 1360; i++)
    {
      rnum = rand();
      rnumtot = rnumtot + rnum;
      data_mat[i+1] = i + 1;
    }

  for (k = 0; k < my_wt; k++)          /* simulates node C */
    for (i = 0; i < 1360; i++)
    {
      rnum = rand();
      rnumtot = rnumtot + rnum;
      data_mat[i+1] = i + 1;
    }

  for (k = 0; k < my_wt; k++)          /* simulates node D */
    for (i = 0; i < 1360; i++)
    {
      rnum = rand();
      rnumtot = rnumtot + rnum;
```

```
        data_mat[i+1] = i + 1;
      }

    pvm_recv(tids[0], me+1);
    pvm_upkdouble(data_mat, disize2*comm_gain, 1);

    pvm_recv(tids[3], me+1);
    pvm_upkdouble(data_mat, disize5*comm_gain, 1);

    pvm_recv(master, me+1);
    pvm_upkdouble(data_mat, disize1*comm_gain, 1);

    pvm_recv(tids[me-1], 20);
    pvm_upkdouble(&go_now, 1, 1);

    if (go_now < 0)
    {
      done = 1;
      pvm_initsend(PvmDataDefault );
      pvm_pkdouble(&go_now, 1, 1);
      pvm_send(tids[3], 4);
    }
  }   /* end of while done == 0 loop */

/* Inform the master I have terminated */
  pvm_initsend(PvmDataDefault );
  pvm_pkint(&me, 1, 1);
  pvm_send(master, 35);

/* Program finished. Exit PVM before stopping */
  pvm_exit();

} /* End of Slave program p4.c */
```

```
/*****************************************************************************
Slave program for scheduled processor 5
*****************************************************************************/
#include "pvm3.h"
#include <stdio.h>
#include <sys/time.h>
#include <time.h>
#include <sys/types.h>
#include <signal.h>
#include <stdlib.h>

    /* CONSTANTS */
#define parent1 2
#define parent2 3

main()
{
  int i,k,mytid, master;
  int tids[20], my_wt, comm_gain;
  int nproc, msgtype, me;
  int rnum, rnumtot=0, done=0;
  double data_mat[10000], go_now;
  FILE *ofp;
  int oldintv = 0, newintv = 0, tcalc;
  struct timeval stime;

/* enroll in pvm */
  mytid = pvm_mytid();
  master = pvm_parent();

  pvm_recv(master, 10 );
  pvm_upkint(&nproc, 1, 1);
  pvm_upkint(tids, nproc, 1);
  pvm_upkint(&my_wt, 1, 1);
  pvm_upkint(&comm_gain, 1, 1);

  pvm_recv(master, 20 );
  pvm_upkint(&disize1, 1, 1);
  pvm_upkint(&disize2, 1, 1);
  pvm_upkint(&disize3, 1, 1);
  pvm_upkint(&dosize2, 1, 1);
  pvm_upkint(&dosize4, 1, 1);

  for ( i=0; i<nproc; i++ )
    if (mytid == tids[i] ) { me = i; break;}

  ofp = fopen("/home3/stone/Thesis/matlab_files/Sched.out","w");
  fprintf(ofp,"\n%d",comm_gain);
  fprintf(ofp,"\n%d",my_wt/4);

  data_mat[0] = 456.33333;
```

```
pvm_recv(tids[2], me+1);
pvm_upkdouble(&go_now, 1, 1);

while (done == 0)
{
  pvm_initsend(PvmDataDefault );
  pvm_pkdouble(data_mat, dosize2*comm_gain, 1);
  pvm_send(tids[0], 1);

  pvm_initsend(PvmDataDefault );
  pvm_pkdouble(data_mat, dosize4*comm_gain, 1);
  pvm_send(tids[2], 3);

  pvm_initsend(PvmDataDefault );
  pvm_pkdouble(&go_now, 1, 1);
  pvm_send(master, me+1);

/**  slave execution core **/
  for (k = 0; k < my_wt; k++)          /* simulates node A */
    for (i = 0; i < 1360; i++)
    {
      rnum = rand();
      rnumtot = rnumtot + rnum;
      data_mat[i+1] = i + 1;
    }

  for (k = 0; k < my_wt; k++)          /* simulates node B */
    for (i = 0; i < 1360; i++)
    {
      rnum = rand();
      rnumtot = rnumtot + rnum;
      data_mat[i+1] = i + 1;
    }

  for (k = 0; k < my_wt*2; k++)        /* simulates node C */
    for (i = 0; i < 1360; i++)
    {
      rnum = rand();
      rnumtot = rnumtot + rnum;
      data_mat[i+1] = i + 1;
    }

  gettimeofday(&stime, (struct timeval*)0);
  newintv=stime.tv_sec*1000000+stime.tv_usec;
  tcalc = newintv-oldintv;

  fprintf(ofp,"\n%d",tcalc);
  oldintv = newintv;

  pvm_recv(tids[0], me+1);
```

```
    pvm_upkdouble(data_mat, disize2*comm_gain, 1);

    pvm_recv(tids[1], me+1);
    pvm_upkdouble(data_mat, disize3*comm_gain, 1);

    pvm_recv(master, me+1);
    pvm_upkdouble(data_mat, disize1*comm_gain, 1);

    pvm_recv(tids[2], me+1);
    pvm_upkdouble(&go_now, 1, 1);

    if (go_now < 0)
      done = 1;

  }    /* end of while done == 0 loop */

  fclose(ofp);

/* Inform the master I have terminated */
  pvm_initsend(PvmDataDefault );
  pvm_pkint(&me, 1, 1);
  pvm_send(master, 35);

/* Program finished. Exit PVM before stopping */

  pvm_exit();

} /* End of Slave program p5.c */

/*     done = pvm_probe(master, done_tag);*/
```

APPENDIX F - HARDWARE MULTICAST NODE PROCESSING PROGRAMS

MASTER PROGRAM (PROCESSOR 1):

```
/*****************************************************************************
Scheduled master program using hardware implemented message multicasts
*****************************************************************************/
#include "pvm3.h"
#include <stdio.h>
#include <sys/time.h>
#include <time.h>
#include <sys/types.h>
#include <stdlib.h>

    /* CONSTANTS */
#define done_lp   1000
#define snl 400

    /* GRAPH TIME VARIABLES */
int  p1out = 300 + 300 + 300 + 300+300;
int  p2out = 350+350 + 350+350 + 300;
int  p3out = 525 + 300+525;
int  p4out = 350 + 400+350 + 350;
int  p5out = 300 + 900;
int  ld_num = 55000;

    /* GLOBAL VARIABLES */
int nproc = 4;
int mytid;                  /* my task id */
int tids[20];               /* slave task ids */
int who, done = 0;
double data_mat[11000], go_now ;

main()
{
  int i, k,rnum, rnumtot;
  int snproc = 3, stids[5], wnl = 0, nl_done = 0;
  struct itimerval tmrval;
  int comm_gain, my_wt;

  printf("\nComm wt =  ");
  scanf("%d", &comm_gain);
  printf("\Ex wt =  ");
  scanf("%d", &my_wt);
  my_wt = my_wt*4;

  printf("\With NET loading type 1, without NET loading type 2:  ");
  scanf("%d", &wnl);

/* initialize matices */
```

```
  for(k=0; k<1500; k++)
    data_mat[k]=(double)k+5.66666;

/* enroll in pvm */
  mytid = pvm_mytid();

/* start up slave tasks */
  pvm_spawn("p2h", NULL, 1, "sun3", 1, &tids[0]);
  pvm_spawn("p3h", NULL, 1, "sun8", 1, &tids[1]);
  pvm_spawn("p4h", NULL, 1, "sun9", 1, &tids[2]);
  pvm_spawn("p5h", NULL, 1, "sun20", 1, &tids[3]);

  pvm_initsend(PvmDataDefault);
  pvm_pkint(&nproc, 1, 1);
  pvm_pkint(tids, nproc, 1);
  pvm_pkint(&my_wt, 1, 1);
  pvm_pkint(&comm_gain, 1, 1);
  pvm_mcast(tids, nproc, 10);

  pvm_initsend(PvmDataDefault );
  pvm_pkint(&p1out, 1, 1);
  pvm_pkint(&p2out, 1, 1);
  pvm_send(tids[0], 20);

  pvm_initsend(PvmDataDefault );
  pvm_pkint(&p2out, 1, 1);
  pvm_pkint(&p3out, 1, 1);
  pvm_send(tids[1], 20);

  pvm_initsend(PvmDataDefault );
  pvm_pkint(&p3out, 1, 1);
  pvm_pkint(&p4out, 1, 1);
  pvm_send(tids[2], 20);

  pvm_initsend(PvmDataDefault );
  pvm_pkint(&p4out, 1, 1);
  pvm_pkint(&p5out, 1, 1);
  pvm_send(tids[3], 20);

  if ( wnl == 1)
  {
    pvm_spawn("s1", NULL, 1, "sun3", 1, &stids[0]);
    pvm_spawn("s2", NULL, 1, "sun20", 1, &stids[1]);
    pvm_spawn("s3", NULL, 1, "sun8", 1, &stids[2]);

    pvm_initsend(PvmDataDefault);
    pvm_pkint(&snproc, 1, 1);
    pvm_pkint(stids, snproc, 1);
    pvm_mcast(stids, snproc, 62);
  }
```

```
/* Begin User Program */

  for (done = 0; done < done_lp; done ++)
  {

    if (done == snl && wnl == 1)
    {
      pvm_initsend(PvmDataDefault);
      pvm_pkint(&ld_num, 1, 1);
      pvm_send(stids[0], 22);
    }

    if (done == done_lp - 1)
      data_mat[0] = -444.555;

    pvm_initsend(PvmDataDefault );
    pvm_pkdouble(data_mat, p1out*comm_gain, 1);
    pvm_send(tids[0], 12);

/* Slave execution cores for P1A, P1B, and P1C */
    for (k = 0; k < my_wt*2; k++)
      for (i = 0; i < 1360; i++)
      {
        rnum = rand();
        rnumtot = rnumtot + rnum;
        data_mat[i+1] = i + 1;
      }

    for (k = 0; k < my_wt; k++)
      for (i = 0; i < 1360; i++)
      {
        rnum = rand();
        rnumtot = rnumtot + rnum;
        data_mat[i+1] = i + 1;
      }

    for (k = 0; k < my_wt; k++)
      for (i = 0; i < 1360; i++)
      {
        rnum = rand();
        rnumtot = rnumtot + rnum;
        data_mat[i+1] = i + 1;
      }

    printf("\nOn loop number %d\n",done);

    if(data_mat[0] >= 0)
    {
      pvm_recv(tids[3], 51);
      pvm_upkdouble(data_mat, p5out*comm_gain, 1);
```

```
    }
  }    /* end of for loop */

  printf("\nThe loop is done\n");

/* Ensure all slaves have quit prior to termination */
  for(i=0; i<nproc; i++)
  {
    pvm_recv(-1, 35);
    pvm_upkint(&who, 1, 1);
  }

 printf("\nProgram m2h.c cwt = %d, ewt = %d is done\n", comm_gain, my_wt/4);

/* Program Finished exit PVM before stopping */
  pvm_exit();

} /** END OF MAIN PROGRAM **/
```

THE INDIVIDUAL SLAVE PROGRAMS:

```
/*****************************************************************************
Slave program for scheduled w/ hardware multicast  for Processor 2
*****************************************************************************/
#include "pvm3.h"
#include <stdio.h>
#include <sys/time.h>
#include <time.h>
#include <sys/types.h>
#include <stdlib.h>

main()
{
    int tids[20],  my_wt, comm_gain;
  int nproc, msgtype, me, i,k,mytid, master;
  int rnum, rnumtot=0, done=0;
  double data_mat[11000];
  int p1out, p2out, p3out, p4out, p5out;

/* enroll in pvm */
  mytid = pvm_mytid();
  master = pvm_parent();

  pvm_recv(master, 10 );
  pvm_upkint(&nproc, 1, 1);
  pvm_upkint(tids, nproc, 1);
  pvm_upkint(&my_wt, 1, 1);
  pvm_upkint(&comm_gain, 1, 1);

  pvm_recv(master, 20 );
  pvm_upkint(&p1out, 1, 1);
  pvm_upkint(&p2out, 1, 1);
  for ( i=0; i<nproc; i++ )
    if (mytid == tids[i] ) { me = i; break;}
  pvm_recv(master, 12);
  pvm_upkdouble(data_mat, p1out*comm_gain, 1);

  while (done == 0)
  {
    pvm_initsend(PvmDataDefault );
    pvm_pkdouble(data_mat, p2out*comm_gain, 1);
    pvm_send(tids[1], 23);

/**  slave execution core **/
    for (k = 0; k < my_wt; k++)          /* simulates node A */
      for (i = 0; i < 1360; i++)
      {
        rnum = rand();
        rnumtot = rnumtot + rnum;
        data_mat[i+1] = i + 1;
```

```
    }
    for (k = 0; k < my_wt; k++)          /* simulates node B */
      for (i = 0; i < 1360; i++)
      {
        rnum = rand();
        rnumtot = rnumtot + rnum;
        data_mat[i+1] = i + 1;
      }
    for (k = 0; k < my_wt; k++)          /* simulates node C */
      for (i = 0; i < 1360; i++)
      {
        rnum = rand();
        rnumtot = rnumtot + rnum;
        data_mat[i+1] = i + 1;
      }
    for (k = 0; k < my_wt; k++)          /* simulates node D */
      for (i = 0; i < 1360; i++)
      {
        rnum = rand();
        rnumtot = rnumtot + rnum;
        data_mat[i+1] = i + 1;
      }
    pvm_recv(master, 12);
    pvm_upkdouble(data_mat, p1out*comm_gain, 1);
    if (data_mat[0] < 0)
    {
      done = 1;
      pvm_initsend(PvmDataDefault );
      pvm_pkdouble(data_mat, p2out*comm_gain, 1);
      pvm_send(tids[1], 23);
    }
  }    /* end of while done == 0 loop */
/* Inform the master I have terminated */
  pvm_initsend(PvmDataDefault );
  pvm_pkint(&me, 1, 1);
  pvm_send(master, 35);

/* Program finished. Exit PVM before stopping */
  pvm_exit();
} /* End of Slave program p2.c */
```

```
/*****************************************************************************
Slave program for scheduled hardware multicast for Processor 3
*****************************************************************************/
#include "pvm3.h"
#include <stdio.h>
#include <sys/time.h>
#include <time.h>
#include <sys/types.h>
#include <signal.h>
#include <stdlib.h>

main()
{
  int i,k,mytid, master, nproc, msgtype, me;
  int tids[20], stids[20], my_wt, comm_gain;
  int rnum, rnumtot=0, done=0;
  double data_mat[11000], go_now = 55.55;
  int p1out, p2out, p3out, p4out, p5out, snproc;

/* enroll in pvm */
  mytid = pvm_mytid();
  master = pvm_parent();

  pvm_recv(master, 10 );
  pvm_upkint(&nproc, 1, 1);
  pvm_upkint(tids, nproc, 1);
  pvm_upkint(&my_wt, 1, 1);
  pvm_upkint(&comm_gain, 1, 1);

  pvm_recv(master, 20 );
  pvm_upkint(&p2out, 1, 1);
  pvm_upkint(&p3out, 1, 1);

  for ( i=0; i<nproc; i++ )
    if (mytid == tids[i] ) { me = i; break;}

  pvm_recv(tids[0], 23);
  pvm_upkdouble(data_mat, p2out*comm_gain, 1);

  while (done == 0)
  {

    pvm_initsend(PvmDataDefault );
    pvm_pkdouble(data_mat, p3out*comm_gain, 1);
    pvm_send(tids[2], 34);

/**  slave execution core **/
    for (k = 0; k < my_wt; k++)          /* simulates node A */
      for (i = 0; i < 1360; i++)
      {
        rnum = rand();
```

```
        rnumtot = rnumtot + rnum;
        data_mat[i+1] = i + 1;
      }
    for (k = 0; k < my_wt; k++)          /* simulates node B */
      for (i = 0; i < 1360; i++)
      {
        rnum = rand();
        rnumtot = rnumtot + rnum;
        data_mat[i+1] = i + 1;
      }
    for (k = 0; k < my_wt*2; k++)    /* simulates node C */
      for (i = 0; i < 1360; i++)
      {
        rnum = rand();
        rnumtot = rnumtot + rnum;
        data_mat[i+1] = i + 1;
      }
    pvm_recv(tids[0], 23);
    pvm_upkdouble(data_mat, p2out*comm_gain, 1);

    if (data_mat[0] < 0)
    {
      done = 1;
      pvm_initsend(PvmDataDefault);
      pvm_pkdouble(data_mat, p3out*comm_gain, 1);
      pvm_send(tids[2], 34);
    }
  }    /* end of while done == 0 loop */

/* Inform the master I have terminated */
  pvm_initsend(PvmDataDefault );
  pvm_pkint(&me, 1, 1);
  pvm_send(master, 35);

/* Program finished. Exit PVM before stopping */
  pvm_exit();

} /* End of Slave program p3h.c */
```

```
/*****************************************************************************
Slave program for scheduled hardware multicast for Processor 4
*****************************************************************************/
#include "pvm3.h"
#include <stdio.h>
#include <sys/time.h>
#include <time.h>
#include <sys/types.h>
#include <signal.h>
#include <stdlib.h>

main()
{
  int i,k,mytid, master, nproc, msgtype, me;
  int tids[20], my_wt, comm_gain;
  int rnum, rnumtot=0, done=0;
  double data_mat[10000], go_now;
    int p3out, p4out;

/* enroll in pvm */
  mytid = pvm_mytid();
  master = pvm_parent();

  pvm_recv(master, 10 );
  pvm_upkint(&nproc, 1, 1);
  pvm_upkint(tids, nproc, 1);
  pvm_upkint(&my_wt, 1, 1);
  pvm_upkint(&comm_gain, 1, 1);

  pvm_recv(master, 20 );
  pvm_upkint(&p3out, 1, 1);
  pvm_upkint(&p4out, 1, 1);

  for ( i=0; i<nproc; i++ )
    if (mytid == tids[i] ) { me = i; break;}

  data_mat[0] = 456.3333;

  pvm_recv(tids[1], 34);
  pvm_upkdouble(data_mat, p3out*comm_gain, 1);

  while (done == 0)
  {
    pvm_initsend(PvmDataDefault );
    pvm_pkdouble(data_mat, p4out*comm_gain, 1);
    pvm_send(tids[3], 45);

    if (data_mat[0] < 0)
      done = 1;

 /**  slave execution core **/
```

```
    for (k = 0; k < my_wt; k++)          /* simulates node A */
      for (i = 0; i < 1360; i++)
      {
        rnum = rand();
        rnumtot = rnumtot + rnum;
        data_mat[i+1] = i + 1;
      }
    for (k = 0; k < my_wt; k++)          /* simulates node B */
      for (i = 0; i < 1360; i++)
      {
        rnum = rand();
        rnumtot = rnumtot + rnum;
        data_mat[i+1] = i + 1;
      }
    for (k = 0; k < my_wt; k++)          /* simulates node C */
      for (i = 0; i < 1360; i++)
      {
        rnum = rand();
        rnumtot = rnumtot + rnum;
        data_mat[i+1] = i + 1;
      }
    for (k = 0; k < my_wt; k++)          /* simulates node D */
      for (i = 0; i < 1360; i++)
      {
        rnum = rand();
        rnumtot = rnumtot + rnum;
        data_mat[i+1] = i + 1;
      }

    pvm_recv(tids[1], 34);
    pvm_upkdouble(data_mat, p3out*comm_gain, 1);

    if (data_mat[0] < 0)
    {
      done = 1;
      pvm_initsend(PvmDataDefault );
      pvm_pkdouble(data_mat, p4out*comm_gain, 1);
      pvm_send(tids[3], 45);
    }
  }    /* end of while done == 0 loop */

/* Inform the master I have terminated */
  pvm_initsend(PvmDataDefault );
  pvm_pkint(&me, 1, 1);
  pvm_send(master, 35);

/* Program finished. Exit PVM before stopping */
  pvm_exit();

} /* End of Slave program p4h.c */
```

```
/*****************************************************************************
Slave program for scheduled hardware multicast for Processor 5
*****************************************************************************/
#include "pvm3.h"
#include <stdio.h>
#include <sys/time.h>
#include <time.h>
#include <sys/types.h>
#include <stdlib.h>

main()
{
  int i,k,mytid, master, nproc, msgtype, me;
  int tids[20], my_wt, comm_gain;
  int rnum, rnumtot=0, done=0;
  double data_mat[10000], go_now;
   int oldintv = 0, newintv = 0, tcalc, p4out, p5out;
  FILE *ofp;
  struct timeval stime;

/* enroll in pvm */
  mytid = pvm_mytid();
  master = pvm_parent();

  pvm_recv(master, 10 );
  pvm_upkint(&nproc, 1, 1);
  pvm_upkint(tids, nproc, 1);
  pvm_upkint(&my_wt, 1, 1);
  pvm_upkint(&comm_gain, 1, 1);

  pvm_recv(master, 20 );
  pvm_upkint(&p4out, 1, 1);
  pvm_upkint(&p5out, 1, 1);

  for ( i=0; i<nproc; i++ )
    if (mytid == tids[i] ) { me = i; break;}
  ofp = fopen("/home3/stone/Thesis/matlab_files/Sched_hs.out","w");
  data_mat[0] = 456.33333;

  pvm_recv(tids[2], 45);
  pvm_upkdouble(data_mat, p4out*comm_gain, 1);

  while (done == 0)
  {
    pvm_initsend(PvmDataDefault );
    pvm_pkdouble(data_mat, p5out*comm_gain, 1);
    pvm_send(master, 51);

/**  slave execution core **/
    for (k = 0; k < my_wt; k++)          /* simulates node A */
      for (i = 0; i < 1360; i++)
```

```
    {
      rnum = rand();
      rnumtot = rnumtot + rnum;
      data_mat[i+1] = i + 1;
    }
  for (k = 0; k < my_wt; k++)          /* simulates node B */
    for (i = 0; i < 1360; i++)
    {
      rnum = rand();
      rnumtot = rnumtot + rnum;
      data_mat[i+1] = i + 1;
    }
  for (k = 0; k < my_wt*2; k++)          /* simulates node C */
    for (i = 0; i < 1360; i++)
    {
      rnum = rand();
      rnumtot = rnumtot + rnum;
      data_mat[i+1] = i + 1;
    }
  gettimeofday(&stime, (struct timeval*)0);
  newintv =stime.tv_sec*1000000+stime.tv_usec;
  tcalc = newintv-oldintv;
  fprintf(ofp,"\n%d",tcalc);
  oldintv = newintv;

  pvm_recv(tids[2], 45);
  pvm_upkdouble(data_mat, p4out*comm_gain, 1);
  if (data_mat[0] < 0)
    done = 1;
}   /* end of while done == 0 loop */

 fclose(ofp);
/* Inform the master I have terminated */
 pvm_initsend(PvmDataDefault );
 pvm_pkint(&me, 1, 1);
 pvm_send(master, 35);
/* Program finished. Exit PVM before stopping */
 pvm_exit();
} /* End of Slave program p5.c */
```

INITIAL DISTRIBUTION LIST

1.	Defense Technical Information Center Cameron Station Alexandria, VA 22304-6145	2
2.	Dudley Knox Library, Code 52 Naval Postgraduate School Monterey, CA 93943-5101	2
3.	Chairman, Code EC Department of Electrical and Computer Engineering Naval Postgraduate School Monterey, CA 93943-5121	1
4.	Prof. Shridhar B. Shukla, Code EC/Sh Department of Electrical and Computer Engineering Naval Postgraduate School Monterey, CA 93943-5121	2
5.	Prof. B. Neta, Code MA/Nd Department of Mathematics Naval Postgraduate School Monterey, CA 93943	2
6.	Prof. Donald A. Danielson, Code MA/Dd Department of Mathematics Naval Postgraduate School Monterey, CA 93943	1
7.	Chairman, Code MA Department of Mathematics Naval Postgraduate School Monterey, CA 93943	1
8.	Dr. Steve Knowles, Code N4/6T Technical Director Naval Space Command Dahlgren, VA 22448-5170	1

9.	Dr. Paul Schumacher, Code 63T Naval Space Command Dahlgren, VA 22448-5170	1
10.	LT. Leon C. Stone, Jr. 414 W. Broadway Princeton, IN 47670	2